CONTENT BASED RETRIEVAL DATABASE MANAGEMENT SYSTEM WITH SUPPORT FOR SIMILARITY SEARCHING AND QUERY REFINEMENT

BY

MICHAEL ORTEGA–BINDERBERGER

M.S., University of Illinois at Urbana Champaign, 1999
Ingeniero, Instituto Tecnológico Autónomo de México, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2002**	2. REPORT TYPE	3. DATES COVERED **00-00-2002 to 00-00-2002**
4. TITLE AND SUBTITLE **Content Based Retrieval Database Management System with Support for Similarity Searching and Query Refinement**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Illinois at Urbana-Champaign,Department of Computer Science,201 N. Goodwin Avenue,Urbana,IL,61802-2302**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**183**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

With the emergence of many application domains that require imprecise similarity based access to information, techniques to support such a retrieval paradigm over database systems have emerged as a critical area of research.

This thesis explores how to enhance database systems with content based search over arbitrary abstract data types in a similarity based framework with query refinement. This scope opens a number of challenges previously not faced by databases, among them:

- Extension of abstract data types to support arbitrary similarity functions and support for query refinement. (Intra type similarity and feedback)
- Extension of the already developed query refinement models under the MARS system to a general multi table relational model. (Inter Type similarity and feedback)
- Extension of query processing models from a set based model where tuples either satisfy or not the query predicate to a result where the degree to which tuples satisfy a predicate is represented by their similarity values. (Similarity predicates)
- Based on the similarity values, return only the best k matches. This implies a sorting on the similarity values and ample optimizations are possible to use lazy evaluation and only compute those answers that the user will see. (Ranked Retrieval)
- Optimization of query execution under the similarity conditions which requires access to specialized indices. Optimized composite predicate merging is possible based on earlier work on the MARS project to compute the similarity value for a predicate based on independent streams rather than using the value directly. (Incremental top-k merging)

We are building a prototype system that implements the proposed functionality in an efficient way and we evaluate the quality of the answers returned to the user.

To my mother.

Acknowledgments

I thank my research advisor, Professor Sharad Mehrotra, for his exceptional guidance during the course of this research. He has been a constant source of motivation and was always available for technical discussions and professional advise.

I want to specially thank Professor Tom Huang for his valuable advise and collaboration that made the MARS project a reality.

I want to thank the members of my dissertation committee, Professors Roy Campbell, Tom Huang and Kevin Chang for valuable advice and encouragement, and the members of my qualifying commitee Professors Roy Campbell, Marianne Winslett and Klara Nahrstedt for ensuring that I was well prepared for my research. I also want to thank the faculty of the University of Illinois at Urbana-Champiagn, and the University of California, Irvine for all I have learned from them.

I thank the members of our research group with whom I worked closely. Among them, I must particularly thank Kaushik Chakrabarti, Yong Rui, Kriegkrai Porkaew, and Ken Hu for being great colleagues and friends.

I wish to thank the Fulbright scholarship commission and their administrative agencies CONACYT, IIE, and USIA for financial support, as well as NSF, NASA and DARPA for funding the Digital Library Initiative Program that helped fund our work.

I am thankful to the staff at Illinois, particularly Barb Cicone and Bonnie Howard from Computer Science, and the staff from OISA for helping me out with all the administrative procedures, sometimes under stressful circumstances.

I thank all my friends, both at Illinois and Irvine for their patience, friendship and comprehension. Particularly I want to thank Ulises Cervantes, Miguel Valdez, Gabriel Lopez, Olga Baos, Seguti, Erick and Javiera Cantu, Fabio and Paula Kon, Carlos Velez, Andrew Hyman, Luis Tavera, Luis Rivera, Enrique Rodarte, Gerardo Valencia, Hugo Coyote, Sergio Carnalla, Hakan, Yueting Zhuang, Iosef Lazaridis, Koushik , Yiming, Yong Rui, Dawit, and the impossible to forget Adolfo Grego.

I owe a special debt of gratitude to my family, specially my Mother, uncle Jaan and aunt Annelies. I would not have been able to get this far without their constant support and encouragement.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DBMS	Database Management System.
df	Document Frequency.
GIF	Graphics Interchange Format Image File Format.
HSV	Hue, Saturation, Value Color Space.
idf	Inverse Document Frequency.
IR	Information Retrieval.
JPEG	Joint Photographic Expert Group Image File Format.
MARS	Multimedia Analysis and Retrieval System.
RGB	Red, Green, Blue Color Space.
SQL	Structured Query Language.
tf	Term Frequency.
UI	User Interface.

Chapter 1

Introduction

Advances in computer technology have redefined search as a fundamental and mission-critical technology. With the massive expansion of the Internet came a radical shift in the demographics of users from tech- savvy, highly trained professionals who understood the technology and its limitations, to naive users seeking "instant gratification". In environments such as web searching, e-business, and data analysis, users increasingly demand better access to information relevant to solve their problems at hand, and can hardly be bothered with high cognitive demands. The search engines that revolutionized the web popularized an easy to use query interface and the ranked retrieval model based on presumed relevance of information that is now deeply entrenched in the users psyche. Today, searching needs are more demanding than ever before.

Huge amounts of critical data remain beyond the reach of advanced search tools. This is not just an availability problem, there is a fundamental disconnect between current technology and the diverse nature of data. Search engines and relational database systems share the common goal of retrieving data associatively: return information that satisfies some criteria regardless of its location. Despite this common goal, search engines typically do not index information stored in databases [145], while databases typically deal only with highly structured data [16]. This divergent evolution has effectively resulted in incompatible tools that unwittingly conspire to hide valuable information to the detriment of users and information providers.

Consider for example the task of finding products matching a user's requirement. Imagine a user Mary who is looking for a 1998 Honda Accord priced around $4000. If she uses a search engine to pose this query, a better 1999 car for $3990 will remain hidden. The fundamental limitation of search engines is that they focus on text and ignore the inherent structure and semantics of the information. Thus, in general, they do not know the meaning of 1998: is it a year?, a house number?, how does it compare to '98? If instead Mary uses a database and she chooses a price range of $3000 to $5000, then not only will a great car with a price of $5001 remain hidden, but she will miss out on the ranked retrieval model she has come to expect. While databases have a rich semantic structure, their fundamental limitations are that they focus on reporting all results that match a query condition exactly. They force a user to convert a vague target value such as "around $4000" into a precise range and therefore suffer from the "near-miss" problem that cause the \$5001 car to be hidden from Mary. This situation illustrates a let down to Mary who expects the best information to aid her decision-making. If Mary truly wants to make an informed decision, she will need to repeatedly restate and submit her search criteria until she has covered her space of interest.

It is no longer acceptable to present users with an adversarial search tool that demands them to sacrifice some search capabilities they have come to expect. We need a search tool that will function against rich unstructured/semi-structured data like web documents (e.g., text, HTML, XML) and multimedia sources (e.g., images, video) in addition to traditional data, and that with a single query, will deliver a single, combined answer and cooperate with the user until the desired information is found.

The key enabling concept for a new-generation of search tools is an integrated similarity search paradigm over structured, semi-structured and unstructured information. A system supporting similarity search returns not only the items that exactly match the search criteria but also those that closely (but partially) match them. The answers are returned sorted/ranked on their "goodness" of match, better matches appearing before the worse ones, thereby increasing the chance of the user finding the most relevant items quickly. In this paradigm, a user can start a query by submitting a partial query with approximate search conditions or by simply selecting an item that closely approximates her information need and requesting for similar items (query-by-example). Users can pose queries via a simple point-and-click, form-based user interface without ever needing to write SQL queries. Similarity searching also allows for meaningful retrieval in presence of inconsistencies and incompleteness of information typically present in heterogeneous data.

A complicating factor when users are searching for information is that they generally do not have a clear mental model of their exact information need at the beginning of the search. As a result, in the context of their exploratory task, users typically have great difficulty posing a query that produces the intended results. This is complicated since the relative importance among different search conditions, and their specific interpretations are generally subjective. This causes them to repeatedly modify their query in an information discovery cycle [80, 154] to probe the database based on certain attributes/features in hopes of improving their results. Hence, user preferences should be taken into account in order to compute results that are relevant to the particular user. The inability to handle user preferences in the exact searching paradigm leads to poor- quality search results.

A promising approach to cope with the difficulty in posing good initial queries, is to assist the users in their information discovery cycle [80, 154] by embracing the concept of *query refinement*. Instead of forcing users to manually modify the query, they can critique the results and and feed back this information to the search tool which then reformulates the query and returns a new set of results. This a form of *relevance feedback* has been successfully applied in the Information Retrieval (IR) literature [137, 8] to textual collections and more recently in the multimedia retrieval literature. We firmly believe this concept, together with similarity based ranking, has a much wider applicability. Consider the following application scenarios:

Example 1.0.1 (Multimedia E-catalog search) *Consider an online garment retailer. The store has an online catalog with many attributes for each garment: the manufacturer, price, item description, and a picture of the garment. We can define similarity functions for each of these attributes, for example, we can define how similar two given prices are, use a text vector model to represent the items description, and can extract color, texture and shape features from the garment image [52]. Through a* query by example *(QBE) user interface, users select or enter a desired price, description, and/or select pictures of attractive garments as color an texture examples. Users then request the "most similar" garments in terms of those attributes. The initial result ranking may not be satisfactory, perhaps the user prefers some colors over others, and dislikes some manufacturers. She expresses her preferences by marking the attributes appropriately, and then re-submits the query. The system computes a new query and returns a new set of answers ranked closer to the users desire.* ■

Example 1.0.2 (Job Openings) *Consider a job marketplace application that contains a listing of job openings (description, salary offered, job location, etc.) and job applications (name, age, resume, home location, etc.) and matches them to each other. The "resume" and "job description" attributes can be are text descriptions, the job and home "location" is a two-dimensional (latitude, longitude) position, and the desired and offered "salary" are numeric data- types. Applicants and job listings are* joined *with a (similarity) condition to obtain the best matches. Unstated preferences in the initial condition may produce an undesirable ranking. A user then points out to the system a few desirable and/or undesirable examples where job location and the applicants home are close (short commute times desired); the system then modifies the condition and produces a new ranking that emphasizes geographic proximity.* ■

We propose a paradigm shift that unites the strengths of Database and Information Retrieval technologies: it brings the similarity searching/ranked retrieval paradigm of search engines into the structured, type-rich access paradigm of databases, thereby providing the best of both worlds in a single integrated system. Unlike search engines, we provide ranked retrieval not just on the text data type, but also on any attribute irrespective of its type. Unlike relational databases, we provide a ranked retrieval of objects based on the degree of match between the object and the query where the degree of match is determined by the subjective interpretation of the user.

In this thesis we propose a new database technology that (1) supports flexible and customizable similarity- based search over arbitrary, application-defined data-types and (2) provides built-in support for query refinement that improves the quality of search results via user interaction with the underlying data. Our approach bridges the gap between the traditional database and search engine technologies; it merges the structured, type rich, database access with the similarity matching/ranked retrieval paradigm of web search engines. The result is a tightly integrated system that provides a flexible and powerful model of data access superior in scope and functionality to that offered by either technology. Furthermore, our approach supports "query refinement" whereby users can personalize their search by providing feedback to the system.

Our suggested approach impacts the design of a database system at all levels. While one way to combine similarity retrieval and query refinement with a database system is to have an application layer on top of a commercial database system, this approach treats the DBMS as little more than storage and is unable to take advantage of advanced features. Instead, a native implementation is free to rethink all aspects of the system and offer the best overall performance.

The query model changes from an exact match to a graded, similarity based matching model that provides a finely grained distinction between answers that is suitable for ranking. This change brings about a fundamental challenge: we now must worry about the quality of the results. Under the exact-match retrieval model, correctness is easy to determine, a record either satisfies the query condition or not. Under a ranked retrieval model, the more important issue is the placement of a record, rather than whether it satisfies the query condition or not. After all, even the worst record satisfies the query condition to some, albeit very low, extent. Once we are faced with considering the quality of the results, considering improvements in the results is a natural extension. Query refinement thus further impacts the database query model by extending the traditional single query-response to a longer-lived session query model [83].

A ranked retrieval model further suggests dramatic changes in query processing: the ordering constraint it imposes also creates an opportunity. The basic assumptions of the exact retrieval model are that all results are equally desirable and must be returned, this results in an eager evaluation approach. Since the number of potential results is anywhere from none to infinity, algorithms have sacrificed economy in the name of speed. Under the ranked retrieval model however, users are interested in the best answers to their query, not all answers. Therefore, the guiding principle should be to find the best answers first while delaying worse results. This shifts the priority from an eager query evaluation to a lazy evaluation strategy that produces on demand only those results the user asks for. The resulting algorithms therefore stress minimizing the work needed to return the next few answers at the expense of less efficiency in returning all answers. The hope is that the overall work will reduce. The impact of query refinement on the query processing algorithms does not conflict with lazy evaluation. The algorithms seek not only to avoid working on results the user has not demanded, but they try to reuse the up front work they did in previous query iterations to reduce the amount of work needed to answer a refined query.

In the remainder of this thesis we focus on each major aspect affected by our approach and highlight it from a user, application developer, or system developer perspective. First we discuss the basic background and related work from the Information Retrieval (IR) and database (DBMS) domains in chapter 2. We then focus on integrating similarity retrieval in the DBMS. Chapter 3 discusses the changes to the relational model that impact the view of an application developer: how queries are formed, how weights are provided, etc. Chapter 4 focuses on the quality of retrieval and discusses how similarity queries are interpreted and the options a system developer must consider. Chapter 5 then turns the system developers attention to algorithms that can implement the similarity ranking model efficiently. After building the similarity retrieval foundation, we turn to the problem of augmenting it with query refinement. Chapter 6 starts by presenting what the application developer can expect from a query refinement system. Chapter 7 then turns to strategies the system developer can use to achieve query refinement while chapter 8 discusses how then can be efficiently implemented. Chapter 9 then presents two case studies from a user perspective. We implemented the garment search application from example X and a user car catalog in our prototype. We discuss how an application developer would build such a system and the interface offered to the user. Finally, we present our conclusions in chapter 10.

Chapter 2

Background and Related Work

This section discusses work that influences the thesis project. Given the interdisciplinary nature of this project, we explore work in several areas: (1) object relational database systems, (2) uncertainty models in databases (3) information retrieval, (4) content based image retrieval, and (5) past efforts to integrate database and information retrieval systems.

2.1 Object Relational Database Model

Databases have evolved in an environment that demands the management of precise data. After several iterations of data models (hierarchical, network), the relational model is considered as one of the most refined database paradigms.[1] Its simplicity and power lies in a small set of concepts that are easily learned, yet the expressive power of queries is immense. The query model is declarative (with a procedural counterpart) [165] that allows for easy specification of the desired result without need of specifying how to compute a result. Powerful query optimization and execution techniques have been developed around this concept.

The relational model is based on a set of relations each one consisting of a set of tuples. A tuple is comprised of a set of fields called attributes where simple atomic data items can be stored. In effect, a relational schema is thus a set of tables with columns of various data types.

To perform queries under this model, two mathematically equivalent paradigms exist:

Relational Calculus is a purely declarative means of specifying the desired result.

Relational Algebra is based on a set of (unary and binary) operators that are applied to tables. It is the procedural equivalent of the relational calculus.

The Structured Query Language (SQL) to support relational systems is largely based on relational calculus, although it incorporates some aspects of relational algebra operators.

[1]The object oriented database model is more expressive than the relational model but has faced considerable hurdles and is not widespread [22].

Given a query expressed in a relational query language, a DBMS determines the answer as a relation consisting of all those tuples that satisfy the predicates with the following relational operators (using the relational algebra approach). The basic relational algebra is very simple:

- $R(a_1, a_2, ..., a_n)$ is a relation over attributes $a_1 \in A_1, a_2 \in A_2, ...a_n \in A_n$ where each A_i is the domain of attribute a_i. The basic operators are:
- The *union* operator which takes two relations R and S with the same number, name and type of attributes and forms a relation T that has tuples that appear in either or both input relations: $T(a_1, a_2, ...a_n) = R(a_1, a_2, ..., a_n) \cup S(a_1, a_2, ...a_n)$.
- Similar to the union operator, the *difference* operator uses two relations R and S over the same attributes and produces a relation T with the tuples of the first relation (R) that are not in the second relation (S): $T(a_1, a_2, ...a_n) = R(a_1, a_2, ..., a_n) - S(a_1, a_2, ...a_n)$.
- The *projection* operator π reduces a relation by discarding certain attributes: $R(a_i, a_j) = \pi_{a_i,a_j}(R(a_1, a_2, ..., a_n))$.
- The *selection* operator $\sigma_{condition}(R)$ which returns only those tuples in R that satisfy the *condition*.
- The *cross product* operator $R \times S$ which creates a new relation T with all the attributes of R and S and forms the cross product of all tuples in R and S: $T(a_1, a_2, ...a_n, b_1, b_2, ...b_m) = R(a_1, a_2, ..., a_n) \times S(b_1, b_2, ...b_m)$
- The *join* operator $R \bowtie_{condition} S$ which is a very frequently used derived operator which combines cross and select and is equivalent to: $\sigma_{condition}(R \times S)$.
- Sorting is not properly a relational operator but we include it here as background for later work on ranked lists. $\rho_{a_i,a_j,...}(R)$ is the sort operator which sorts the tuples in a relation based on the combined attributes $a_i, a_j, ...$.

A subset of the relational algebra operators is *complete* if it can express any relational algebra operation. The set of operations: $\{\cup, -, \pi, \sigma, \times\}$ is complete for the basic relational model and operators. Other derived operators exist which are combinations of several of the basic operators such as the *division*. Yet other relational operators such as *outer joins* are not expressible by the basic operators since they do not conform to basic relational assumptions and were created for practical considerations. We will ignore these extra operators.

The basic relational model requires tables to be in the first normal form [36] where every attribute is atomic. This poses serious limitations in supporting applications that deal with objects/data types with rich internal structure. The only recourse in the relational model is to map the complex structure of the applications data types from and to the relational model every time the object is read or written from or to the database. This results in extensive overhead making the relational approach unsuitable for advanced applications that require support for complex data types.

The above limitations of the relational systems resulted in much research and commercial development to extend the database functionality with rich user-defined data types in order to accommodate the needs of advanced applications. Research in extending the relational database technology occurred along two parallel directions.

The first approach, referred to as the object-oriented database (OODBMS) approach attempted to enrich object-oriented languages such as C++ and Smalltalk with the desirable features of databases, such as concurrency control, recovery, and security. The OODBMS approach strived to support the rich data types and semantics of object oriented languages with minimal changes to add additional features such as persistence and transactions. Examples of systems that followed this approach include research prototypes such as [23] and a number of commercial products [9, 92].

The Object-relational database (ORDBMS) systems, on the other hand, approached the problem of adding additional data types differently. ORDBMSs attempted to extend the existing relational model with a full-blown type hierarchy of object-oriented languages. The key observation was that simple data types had been used thus far as a consequence of the market and limited computing resources and that the concept of the domain of an attribute need not be restricted to simple data types. Given its foundation in the relational model, the ORDBMS approach can be considered a less radical evolution compared to the OODBMS approach. The ORDBMS approach produced such research prototypes as Postgres [155], and Starburst [67] and commercial products such as Illustra [156]. The ORDBMS technology has now been embraced by all major vendors including Informix [77], IBM DB2 [28], Oracle [112], Sybase [159], and UniSQL [84] among others. The ORDBMS model has been incorporated in the SQL-3 standards.

While OODBMSs provide the full power of an object oriented language, they have lost ground to ORDBMSs. Interested readers are referred to [156, 22] for a good insight from both a technical as well as commercial perspective for this development. In the remainder of this work, we will concentrate on the ORDBMS approach.

The object-relational model retains the concept of tables and columns in tables from the relational model. Besides the basic types available, it augments the possible data types allowed in columns to user-defined *abstract data types* (ADTs) and various collections of these and basic types. The ADTs can conceivably be any type and typically consist of a number of data fields and functions that operate on them. These functions, written by the user, are known as *User Defined Functions* (UDFs) and are equivalent to *methods* in the object oriented context. In the object-relational model, the fields of a table may correspond to basic DBMS data types, other ADTs or even just some storage space whose interpretation is left to the user defined methods for the type [77]. The following example illustrates how a user may create an ADT and include it in a table definition:

```
create type ImageInfoType ( date varchar(12) ,
                            location_latitude real ,
                            location_longitude real )

create table SurveyPhotos ( photo_id integer primary key not null,
                            photographer varchar(50) not null,
                            photo_location ImageInfoType not null,
                            photo blob not null)
```

The type *ImageInfoType* stores the location at which a photograph was taken together with the date stored as a string. This can be useful for nature survey applications where a biologist may desire to attach a geographic location and a date to a photograph. This type is then used to create a table with an id for the photograph, the photographers name, the photograph itself (stored as a binary large object or BLOB) and the location and date when it was taken.

The ORDBMSs extends the basic SQL to allow user defined functions (once it has been compiled and registered with the DBMS) to be called directly from within SQL queries thereby providing a natural mechanism to develop domain specific extensions to databases. The following example shows a sample query that calls a user-defined function over the type declared above:

```
select photographer, convert_to_grayscale(photo)
       from SurveyPhotos
       where within_distance(photo_location,'1', '30.45, -127.0')
```

This query returns the photographer and a gray scale version of the image stored in the table. The *photo_location* UDF is a predicate that returns true if the place where the image was shot is within 1 mile of the given location. Note that the UDF *within_distance* is a predicate that returns true for locations closer than one mile and false otherwise. This UDF ignores the date the picture was taken, demonstrating how predicates are free to implement any semantically significant properties of an application. Also note that the UDF to convert the image to gray-scale is applied to a returned attribute. The last significant ability of ADTs is the support of inheritance of types and as a consequence, polymorphism. This does introduce some problems in the storage of ADTs as it complicates the existing storage mangers that assume all rows in a table share the same structure which is no longer true. Several strategies were developed to cope with this problem [53] including dynamic interpretation and a number of real tables one for each possible type that correspond to a larger virtual table that forms the user defined table.

When a DBMS receives an SQL query, it first validates the query and then determines the strategy to evaluate it. Such a strategy is called the query evaluation plan or simply *plan* and is represented using an operator tree [57]. For a given query, there are usually several different plans that will produce the same result; they only differ in the amount of resources needed to compute the result. The resources include time and memory space in both disk and main memory. The query optimizer first generates a variety of plans by choosing different orders among the operators in the operator tree and choosing different algorithms to implement these operators and then selects the best plan based on the available resources [57]. The two common strategies to compute optimized plans are (1) *rule-based* optimization [67] and (2) *cost-based* optimization [140]. In the rule based approach, a number of heuristics are encoded in the form of production rules that can be used to transform the query tree into an equivalent tree that is more efficient to execute. An example rule is pushing down selections below joins as it reduces the sizes of the input relations and hence the cost of the join operation. In the cost based approach, the optimizer first generates several plans that would correctly compute the answers to a query and computes a cost estimate for each plan using the system maintained statistics for each relation (i.e., number of tuples, number of disk pages occupied by the relation etc.) as well as for each index (i.e., number of distinct keys, number of pages etc.). Subsequently, the optimizer chooses the plan with the lowest estimated cost [140].

Performance evaluation and optimization in traditional database systems is only considered in the temporal (computational) sense. Since determining the quality of the answers is trivial given the precise retrieval model — they are either correct or not — and therefore there is little need to evaluate the quality of the results. The TPC [163] benchmarks are the standard benchmarks used in this environment and are focused on computational performance. Query optimization therefore is geared towards reducing the computational effort required to perform a query.

2.2 Uncertainty Models in Databases

Emerging applications pose an increasing demand on DBMSs to store and process imprecise information alongside precise information traditionally stored in databases. There are a number of sources for imperfect and imprecise data and different ways to handle such problems. A good survey appears in [118], but here we focus on the models that are more applicable to Content Based Retrieval applications. Such applications require many important extensions to existing DBMS technologies including: (1) mechanisms to represent uncertainty in stored data, (2) extensions to DBMS query languages to support uncertain queries, and (3) mechanisms to process uncertain queries. Uncertainty manifests itself in data and in queries:

Uncertainty in Stored Data: Uncertainty in data may arise due to a variety of reasons from multiple different sources. For example, in an image database, an image is represented as a collection of visual features (e.g., color, texture, shape, etc.). These features taken together form an imprecise representation of the image content. In a spatio-temporal database, the location of an object may be uncertain due to the limited precision of the sensor tracking the object and the temporal latency between successive readings.

Uncertainty in Queries: Similar to the descriptional uncertainty in stored data, uncertain queries may arise due to a variety of reasons. For example, the user may lack the capability or find it cumbersome to express his/her information need as a precise query over the stored data. Even when the user has a precise information need, he/she is unable to specify the query using the imperfect data that is used to model the objects in the database.

Using the Object Relational Model, it is possible to assign a score between a query object and objects in the database. In such an implementation, the user maps her query into SQL and user-defined functions implement the imprecise operations (e.g., match between images, precision of moving objects) and associate a score with each object retrieved. The set of retrieved objects are sorted based on the application-specific criterion. While the ORDBMS model supports UDFs, the relational operators are not aware of the scoring and ranking implied in the results. Done in this way, imprecision in the processing *does not* permeate database query processing.

Among the many models [118] that deal with incomplete and imprecise information in databases, we focus on two of the best known paradigms recognized as useful for representing uncertainty in databases [40, 172, 115]. One is based on fuzzy logic and the other based on probability theory.

Fuzzy Logic for Databases. Fuzzy logic [170, 171] has been proposed for use in the context of the relational data model [128]. Fuzzy logic rests on the recognition that many real world facts cannot be precisely stated. Often we use subjective values to qualify subjects, "tall" could be such an imprecise qualifier. The actual value for such an attribute cannot be determined, as it depends on contextual information.

Under this model, attributes can have a fuzzy number associated with them, or a set of numbers representing a fuzzy set. For Content Based Retrieval the most natural interpretation is to assign score values in the range $[0, 1]$ with 1 as the best score to the result of the match of a tuple with the query [115].

Relational operators have to be suitably modified to account for conditions on these values [128]. The most common interpretations for combining scores under this model are: (1) the maximum of the scores for the *union* or *or* operators, (2) the minimum of the scores for the *intersection* or *and* operators, and (3) one minus the score for the *negation* operator. Other numeric interpretations are possible for these operators, [118] presents more details on the different approaches.

Probability Models for Databases. The probabilistic model is firmly rooted in probability theory and under this interpretation the score of the match between an object and the query is considered to be the probability that the object matched the query [115], assuming the score is in the range $[0, 1]$. Then, probability theory is used to combine the values depending on the operators involved. A key assumption which may not always hold, specially in Content Based Retrieval applications is that of independence. Independence simplifies the computations, and is based on the assumption that the different features extracted for objects are independent of each other, i.e. the color content of an image is not at all related to the texture.

Under probabilistic database models, a probability is assigned to a tuple, this can be either a number [10, 40, 91] or an interval [1]. By using an interval some of the problems introduced by the assumption of independence can be resolved. The most straightforward method to combine scores under this model are: (1) the sum of the probability values minus its multiplication when two probability values are involved for the *union* or *or* operators, (2) the multiplication of probability values for the *intersection* or *and* operators, and (3) one minus the probability value for the *negation* operator. References [118, 172] present more details on the different approaches.

VAGUE [106] presents a model that lets users choose the interpretation of a predicate before executing a query, it does not allow the user to change the interpretation of query predicates after the fact. Similarly, no uncertainty model in the literature [118] accounts for user subjectivity and desire to iteratively change the interpretation of operators to make the results more appealing.

2.3 Information Retrieval

Information Retrieval (IR) is the field that historically has dealt with the search of textual documents based on a users information need expressed in textual form, IR systems typically have a large collection of textual documents from which all those documents that match a users query are returned. This section gives a background on modern IR [137, 8]. First, we discuss document models used in IR and the criteria used to compute the similarity between documents. Then we review query refinement techniques used in IR.

2.3.1 Information Retrieval Models

In an IR system, a document is represented as a collection of features (also referred to as terms). Examples of features include words in a document, citations, bibliographic references, etc. Although features and query models are commonly presented together in the literature, the features extracted and the way they are obtained is largely independent of the query model, however the features obtained need to sustain the query model. The most common features used are the words in the document themselves. Many variations and refinements to compute these features have been developed over time, some refinements include:

- Ignore case. This compares words and ignores whether they are in upper or lower case.
- Stop-words. Use a list of words that are too common to be meaningful such as prepositions and connectors.
- Stemming. Reduce words to their original state, folding for example the plural and singular forms of a word together. This reduces the number of words that are indexed and also allows for easier word to word comparisons, i.e. child and childish will be considered equal.
- Thesauri. This allows systems to consider synonyms so that the total number of words is less and also to gives more flexibility to users in posing queries.

Beyond these enhancements, many other more sophisticated enhancements exist such as using noun-phrases instead of single words in hopes that phrases will contain more semantic meaning. Another enhancement that is tied more to a specific query paradigm is that of Latent Semantic

Indexing [8], where many words are folded into smaller categories based on mathematical principles in hopes that this automatic method will detect when two words are synonyms and consider them equal. Indexing of other features such as titles, authors or citations follows similar principles except these features are stored separately.

A user specifies her information need to the system in the form of a query. Given a representation of the user's information need and a document collection, the IR system estimates the likelihood that a given document matches the user's information need. The representation of documents and queries, and the metrics used to compute the similarity among them constitute the *retrieval model* of the system. Existing retrieval models can be broadly classified into the following categories:

Boolean Models Traditionally, commercial IR systems have used the Boolean model. Systems based on Boolean retrieval partition the set of documents into either being relevant or not relevant and do not provide any estimate as to the relative importance of documents in a partition to the user's information need. The Boolean model has also been extended to allow for ranked retrieval in the text domain (e.g. the *p*-norm model [136]), and in the image retrieval domain (see our previous work in [115]).

Let $\{r_1, r_2, \ldots, r_k\}$ be the set of terms in a collection. Each document is represented as a binary-valued vector of length k where the i^{th} element of the vector is assigned *true* if r_i is assigned to the document. All elements corresponding to features/terms not assigned to a document are set to *false*. A query is a Boolean expression in which operands are terms. A document whose set of terms satisfies the Boolean expression is deemed to be relevant to the user and all other documents are considered not relevant.

Vector-based Models In the vector space model, a textual document is considered as a collection of words. The same word can appear multiple times in the same document and thus the notion of *term frequency* (tf) arises. Term frequency is the number of times a word or term appears in a document, the higher the count, the more important the word is to the document. Conversely, a word can appear in many documents, the *document frequency* (*df*) is the number of documents in which the word appears at least once. A high document frequency indicates that a term has low discrimination value among documents and thus is of little value for retrieval. All terms in a document are then assigned weights based on a combination of the term frequency and the document frequency of the term. To denote the penalty for a high document frequency, this figure is usually inverted and named *inverse document frequency* (*idf*), and is computed as: $idf = \log \frac{collection\ size}{df}$.

Experiments have shown that the product of tf and idf is a good estimation of the weights [19, 137, 144]. The final weight w_i for term i in the document is: $w_i = tf_i \times idf_i$ Now, each document has a weight assigned for all possible words (words not in the document have a weight of 0). If there are N_{word} distinct words in the collection of all documents, then each document can now be viewed as a point in an N_{word} dimensional space.

A query is represented in the same way as a document, weights are extracted for all the terms in the query and then documents "close" to the query point are searched for. To determine which documents are close, or similar to a query, a similarity function is defined over the domain of a pair of documents. The resulting value of applying this function to any two documents, or a document and a query represented as a document, is a measure of the similarity between the documents. A similarity of 1 indicates they are identical, and a value of 0 indicates they are not similar at all. Many similarity measures between a document and the query document have been proposed [137], the most common being the cosine of the angle between the document and the query vectors in the N_{word} dimensional space [137] and the origin. To compute the cosine of the angle between the two vectors, we take advantage of the following equivalence:

$$D \cdot Q = ||D|| \ \ ||Q|| \ \ \cos \ \theta \tag{2.1}$$

By rearranging 2.1, the similarity between the query and document vectors is computed as:

$$similarity(D, Q) = \frac{D \cdot Q}{||D|| \ ||Q||} \tag{2.2}$$

$D \cdot Q$ denotes the inner product of the query and document vectors defined as:

$$D \cdot Q = \sum_{i=1}^{i=N_{word}} D_i \times Q_i \tag{2.3}$$

$||D||$ and $||Q||$ denote the length or norm-2 of the vectors defined as:

$$||Vector|| = \sqrt{\sum_{i=1}^{i=N_{word}} Vector_i^2} \tag{2.4}$$

Vector models are suited for situations where all terms used to describe the document content are of the same type, i.e. they are homogeneous.

Probabilistic Retrieval Models In these models the system estimates the probability of relevance of a document to the user's information need specified as a query. Documents are ranked in decreasing order of probability relevance estimate. Given a document and a query, the system computes $P(R|d, q)$ which represents the probability that the document d will be deemed relevant to the user's information need expressed as the query q. These probabilities are computed and used to rank the documents using Bayes' theorem and a set of independence assumptions about the distribution of terms in the documents. An example of this category is the INQUERY system [20].

Weighted Summation Models Let $T = \{t_1, t_2, \ldots, t_n\}$ be the list of terms associated to a document D_i and $S = \{s_{1,i}, s_{2,i}, \dots, s_{n,i}\}$ the list of similarity values for each term of document D_i to the corresponding term in the query object Q. That is, $s_{j,i}[Q, D_i] = similarity(t_j(Q), t_j(D_i))$. In weighted summation, the individual component-wise similarities are combined by attaching a weight to each similarity value and adding them together, according to $S[Q, D_i] = \sum_{j=1}^{j=n} weight_j \times similarity(t_j(Q), t_j(D_i))$ where $\sum_{j=1}^{j=n} weight_j = 1$. This model is similar to the vector model in its structure. It is more flexible than the vector model in that the similarity between two terms is computed by an arbitrary similarity function which allows the merging of heterogeneous terms or features.

Latent Semantic Indexing This family of models closely follows the vector model. The matrix formed by all the words and documents is then reduced in dimensionality via Eigen value analysis. Words that frequently co-occur will be folded into the same dimension, thus creating a semantic unit. The many proposed algorithms vary in the way the dimensionality is reduced and the distance function between the resulting lower dimensional vectors.

To compare between different methods, their retrieval performance on standard benchmarks is compared. The typical benchmark datasets are the TREC collections [110]. Performance in this domain is qualitative and is measured differently from database systems (execution speed). Performance in this domain is measured with two values, precision and recall. Given a query and an answer consisting of a set of documents, the precision is the ratio of (subjective) documents considered relevant to the query vs. the number of returned documents. A perfect precision of 1 is obtain when every returned document is relevant to the question. The second measure is recall. This indicates the ratio of relevant documents to the query that were retrieved vs. all relevant documents in the collection. A perfect recall of 1 means that all relevant documents were indeed retrieved. These two measures conflict. For once, perfect recall can be obtained by returning the whole collection at a substantially low precision. Usually, precision recall curves are plotted to measure a systems performance.

IR techniques are the driving force behind Internet search engines such as Altavista, Lycos, Hotbot, Inktomi, Google and other search engine products. Many commercial systems, are not very feature rich and trade simplicity for speed and scalability, yet their success is evident by their widespread use.

2.3.2 Query Refinement

IR query models compute the relevance of each document to the users query and then typically output a ranked list of answers. The computation of results is done according to the selected models in a straightforward way.

The user may receive some good and some bad results, to enhance the retrieval performance, the process of query refinement can be used. Query refinement consists of changing the original query to improve its results. One way to perform query refinement is to present the user the results and ask her to state which are relevant and which are not; thus query refinement or relevance feedback is the process of automatically adjusting an existing query using information fed-back by users about the relevance of previously retrieved documents.

Query refinement methods change a query and are dependent on the retrieval method chosen. There exist query refinement techniques for many methods, including the Boolean, vector, probabilistic, and LSI models.

In the Boolean model, feedback usually takes the form of adding new conjuncts to the query.

The vector model has a widely used and simple relevance feedback mechanism. In the vector model, a query is a point in a large dimensional space, the point can be moved to reflect the fact that the users interest is in a different region of space. If the query point is p, and the set of relevant points (vectors) is R and the set of non relevant points is NR, then the optimal query for the same similarity function is [19, 137, 144]: $p_{opt} = p + \sum_{r \in R} \frac{r}{|R|} - \sum_{nr \in NR} \frac{nr}{|NR|}$ where $|R|$ and $|NR|$ are the number of points in the relevant and non relevant sets. In general R and NR are not known, but are approximated by the users judgment of the return set. The above technique is applied to arrive at a better approximation of the optimal query. By iteratively repeating this procedure we asymptotically approach p_{opt}. In addition to shifting the location of the query point, it is also possible to change the weights in each dimension thus changing the shape of the query area (that is, the retrieval model used to compute similarity between objects). A different approach presented in [125] is to use multiple points to compute a query result instead of a single one. This allows to form irregularly shaped query regions and has proven to give better feedback performance than query point movement at a low increase in computational cost. The approach presented in [169] follows a multi-point query strategy with an aggregation function that allows for concave query regions that can even represent query regions with “holes”.

Probabilistic models base their relevance feedback in propagating probability changes through i.e. a Bayesian network such as used in INQUERY [20].

In weighted summation queries, the similarities of query to object features are combined through weighted summation, and then propagated up the query tree through the same mechanism. In this relevance feedback process, the individual weights are changed to better reflect the users information need. Without modifying the structure of the query tree, the individual weights are updated. To update the weights, the result lists returned for each node in the query tree are cut off at a certain similarity value, then all the relevant objects indicated by the user are searched in these lists. Many strategies exist to select the new weight for each node that produced a list including: counting the number of relevant objects in the list, minimum similarity value among all objects in the list, and sum the similarity values for all relevant objects in the list [123]. Naturally, this process does not yield weights that add up to 1. The final step is to normalize the weights so that they add up to 1 for all children of a node. Once this process is complete, the query is re–executed and shown to the user for another iteration of relevance feedback.

2.4 Image Retrieval

With the dramatic increase in computational power and software flexibility, other types of data beyond text have become increasingly more important such as images, time-series, etc. which now demand retrieval techniques similar to those provided by traditional text IR systems. The retrieval paradigm offered by IR systems has been termed content-based retrieval to reflect the nature of the indexing which centers on the semantics of the data rather than the data itself. To this end, much research has been conducted to extend the IR techniques to other domains.

A number of image retrieval systems exist that have adapted IR techniques. One such system is our MARS [75] system, where suitably adapted Boolean [115], vector and summation models [134, 123] are used for image based retrieval. Within the context of the MARS system, we developed a number of query refinement techniques to complete the image retrieval process [134, 125, 123].

Existing approaches to multimedia information retrieval belong to one of the following two categories. The first approach, followed by many existing multimedia database systems, is to annotate multimedia data with text [127, 152, 51, 81], and then use existing textual information retrieval systems [137, 85, 88, 50] to search for multimedia information indirectly using the annotations. Due to the difficulty in capturing multimedia content using textual annotations, non-scalability of the approach to large data sets (due to the high degree of manual effort required in producing the annotations), and the high degree of subjectivity of the annotations to the human indexer, the second approach in which visual features are directly used for retrieval has emerged. Examples of such an approach include commercial products like *Query By Image Content* (QBIC) developed at the IBM Almaden Research Center [48, 52], the *Virage* system developed by Virage Technologies Inc. [7], and the *Visual RetrievalWare* developed by Excalibur Technologies. Related research prototypes being developed in academia include the PhotoBook project at MIT [119, 121, 120], Alexandria project at UC, Santa Barbara [100], VisualSeek [147], Advent system at Columbia University [150], and the Chabot project at UC, Berkeley [111].

Similar to MARS, the objective of these systems is to support content-based retrieval over images and multimedia objects. The focus of these efforts has been on techniques to efficiently extract and represent low-level visual features from images and multimedia objects based on which users may wish to retrieve multimedia information. For example, the QBIC, Virage and Visual RetrievalWare systems have developed powerful representations for lower-level image features: color, texture, shape, appearance, and layout. Research in the Advent system has explored mechanisms for extracting color and texture features in both compressed as well as uncompressed images. While this research has resulted in many significant contributions in feature extraction and representation, the retrieval mechanisms supported by these systems have, so far, been comparatively primitive. Typically, users retrieve objects directly based on their low-level feature values (e.g., retrieve images that contain a lot of "blue"), or using an *adhoc* combination of features via user-specified weights (e.g., retrieve images similar in color and texture to a given image, where weight of color is 0.8 and that of texture is 0.2).

All systems model an image as a set of features, for example the color and texture features and for each feature a set of representations, for example color histogram and color moments for the color feature (see below). For each feature a function is used to compute the degree to which two instances of the feature representation match. The matching functions can be based on a distance measure where a distance of 0 between two features means they match completely, and values above 0 indicate reduced match, or they can be based on a similarity match where 1 means perfect match and as values go to 0, the quality of the match decreases. We make no specific assumption either way other than requiring that all functions be consistent in that they all return a score which can be either distance or similarity based, but not mixed.

To determine the degree of match between two images, some combination of the individual feature matches is used. Typically this takes the form of a Boolean algebra with fuzzy or probabilistic interpretations [115] or a weighted summation model where the final score is a sum of the individual feature to feature match between a query and an image [123].

The retrieval performance of an image database is inherently limited by the nature and the quality of the features used to represent the image content. We briefly describe some image features. Detailed discussion on the rationale and the quality of the features can be found in references [49, 160, 102, 105, 135].

Color Features: The color feature is one of the most widely used visual features in image retrieval. Many approaches to color representation, such as color histogram [158], color moments [157], color sets [149], have been proposed in the past few years.

Texture Features: Texture refers to the visual patterns that have properties of homogeneity that do not result from the presence of only a single color or intensity [151]. It is an innate property of virtually all surfaces, including clouds, trees, bricks, hair, fabric, etc. Texture contains important information about the structural arrangement of surfaces and their relationship to the surrounding environment [68]. Because of its importance and usefulness in Pattern Recognition and Computer Vision, extensive research has been conducted on texture representation in the past three decades, including the co-occurrence matrix based representation [68], Tamura texture representation [161], and wavelet based representation [148, 30, 90, 63, 89, 162]. Many research results have shown that the wavelet based texture representation achieves good performance in texture classification [148].

Shape Features: Shape of an object in an image is represented by its boundary. A technique for storing the boundary of an object using a modified Fourier descriptor (MFD) is described in [135]. The Euclidean distance can be used to measure similarity between two shapes, however [135] proposes a similarity measure based on standard deviation that performs significantly better compared to the simple Euclidean distance. The representation and similarity measure provide invariance to translation, rotation, and scaling of shapes, as well as the starting point used in defining the boundary sequence. A large number of shape representations exist including Chamfer shape descriptor [133] as well as our own adaptive shape representation [26].

Color Layout Features: Although the global color feature is simple to calculate and can provide reasonable discriminating power in retrieval, it tends to give too many false alarms when the image collection is large. Many research results suggest that using color layout (both color feature and spatial relations) is a better solution. Systems that use color layout include Blobworld [24] and VisualSeek [147].

Depending on the extracted feature, some normalization may be needed [115, 134]. The normalization process serves two purposes: (1) It puts an equal emphasis on each feature element within a feature vector. To see the importance of this, notice that in the texture representation, the feature elements may be different physical quantities. Their magnitudes can vary drastically, thereby biasing the distance measure. This is overcome by the process of *intra-feature* normalization. (2) It maps the distance values of the query from each atomic feature into the range [0,1] so that they can be interpreted as the degree of membership in the fuzzy model or relevance probability in the probabilistic model, or just a similarity score for the weighted summation model. While some similarity functions naturally return a value in the range of [0, 1], e.g. the color histogram intersection; others do not, e.g. the Euclidean distance used in texture. In the latter case the distances need to be converted to the range of [0, 1] before they can be used. This is referred to as *inter-feature normalization*.

2.5 Integration of Information Retrieval with Databases

There is a body of work relating to the integration of IR and DBMS systems [73, 94, 38, 54, 64, 93, 153, 166]. Their motivation for exploring such an integration is a result of increasing requirements for IR systems to support (1) access to information (e.g., SGML documents) by not only their content, but also their structure. (2) online insertion, deletion and retrieval of documents. Traditionally, IR systems have ignored document structure in retrieval and have only supported off-line creation and modification of indices. The work can be divided into two categories, data models developed for IR integration based on probabilistic data models and experimental system integration resulting in prototypes.

2.5.1 Model Based Integration

From the theoretical perspective, database and information retrieval systems can be characterized according to table 2.1 [93]. The last line in table 2.1 was not taken from the references, instead we feel that this is a key distinction that is so far missing in the literature and is integral to our approach: SQL queries are stateless, while we introduce a *session query*, in which an original query is refined over multiple iterations.

Few theoretical frameworks for integration of IR and DBMSs exist. Those that exist propose some imprecise relational data model that may support an IR system layered on top of it. Consequently, none of these models provide features such as query refinement or descriptions to perform query execution.

Fuhr proposed a model for the integration of IR and DB systems [54, 55, 56]. The model presents a probabilistic relational algebra that it based on intentional semantics. The benefit of the

Table 2.1: Comparison between Information Retrieval and Data Retrieval Approaches to Querying

	Concept	Information Retrieval	Data Retrieval
1	Retrieval Models	Probabilistic	Determinate
2	Indexing	Derived form contents	Complete Items
3	Matching/Retrieval	Partial or "best" match	Exact match
4	Query types	Natural language	Structured
5	Results criteria	Relevance	Any match
6	Results ordering	Ranked	Arbitrary
7	*Query duration*	*Iterative*	*Stateless*

approach is to provide a relational database schema that can store the degree to which a textual term relates to a document. The limitations of his approach include:

- No specification how to process queries and do optimization.
- There is no support for query refinement.
- The storage of data is more akin to an index than a data type and thus requires specific operations.
- Text terms are handled but no other data-types are supported, and the algebra to extend relational operators is customized to the textual representation.

A fundamental problem with Fuhrs approach is that the information stored in relational tables corresponds to the indexing done in IR systems thus providing the wrong level of abstraction. It is more natural to think of the term information in the context a database index. Fuhrs approach can be characterized to be in the *extended RDBMS approach* described below, however he does not consider general purpose data types.

In addition to models explicitly attempting the integration of IR support into DBMSs, there are modified relational models to incorporate uncertainty.

2.5.2 Experimental System Integration

In contrast to theoretical approaches, more practical implementations exist. The general consensus in these projects is that building a fully integrated IR–DBMS system is too much work. Rather than do that, they rely on some already existing DBMS (relational or object oriented) and in some way either layer or use a parallel IR system. From a practical point of view, [73] presented a five level hierarchy for integrating DB and IR systems. This hierarchy is:

The standard RDBMS approach uses a standard RDBMS to store IR data in it. No facilities particular to IR are provided in the DBMS but need to be layered on top with an application program.

The extended RDBMS approach extends a RDBMS with IR type data-types (text) and some functions on them (regular expression matches). Examples of this approach are the Informix Datablades, DB2 Extenders and Oracle Cartridges.

The extended IR approach adds some relational capability to an IR system.

External integration approach in which IR and DBMS systems coexist in the same environment with a common query interface. The interface separates the query into the IR and DBMS parts and send each to the appropriate system and then merges the results.

The full integration approach combines the functionality of both systems into a hybrid DBMS capable of managing IR and traditional DBMS data.

In an early attempt, [94] suggests the *standard RDBMS* approach although they acknowledge its limitations. Storing document terms in a straightforward manner in an RDBMS is slow for execution and does not provide for any advanced retrieval techniques.

Another early attempt is described in [38]. This attempt corresponds to the *standard RDBMS* interpretation, with some description of how to layer IR functions on top.

A much more advanced system is OCTOPUS used by the Dutch police [74]. The basic data management is conducted on structured (exact) data, however several functions supporting less structured data (text) are supported. Ranking and query refinement are not supported, but some desirable data integrity features are (transactions).

The HYDRA hybrid DB and IR system by the Integrated Publication and Information Systems Institute is a combination of full and external integration [64]. In reality, what it does is to store common (text) IR inverted files in INQUERY [20]. Queries are posed via a proprietary query facility (form), that separates the exact from the probabilistic predicates, the IR part of the query is sent to INQUERY the exact part is sent to Sybase. Results are then stored in a temporary table, pruned with a threshold and combined via SQL with exact matching data. This approach is clearly poor as it completely ignores issues of efficient query execution by the simple fact that the RDBMS and INQUERY really do not know about each others functions. Furthermore, the problems of replicated and consistent data arises.

The GIPSY geographic IR system focuses on applying IR techniques to the geographic domain [93]. GIPSY tries to integrate IR text and spatial (fuzzy) matching with some exact data (building locations, etc.). GIPSY follows a proprietary implementation where a DBMS is not considered, exact data is obtained from flat files.

The FIRE approach [153] uses an OODBMS (ObjectStore) to implement an IR system on top of it. In some sense this corresponds to the *standard DBMS* integration approach and can be considered to be simple. The authors claim they can rely on the OODBMS to perform optimization, however, the DBMS does not know about the IR operations and these need to be included as part of a users application code.

In [166], two integration attempts of the INQUERY text IR system with an RDBMS are presented. The first attempt users the DEC RDB DBMS to store the INQUERY indexes in it, effectively using RDB as the underlying file system from which INQUERY obtains its data. The second, more integrated approach again stores the inverted files into the RDBMS but uses functions that implement the basic INQUERY operators (PAND, POR, ...) in the where clause. The DBMS does not understand the operation being performed and has trouble optimizing for it. For optimization, the authors rely on a *cooperative index* that should be but is not integrated into the DBMS engine. This is probably the most complete system integration attempt in the literature.

The extended integration approach is used currently by most commercial database systems through their object relational capabilities. This approach has the advantage of simplicity with its primary limitations including issues of efficiency, ad hoc semantics (it is not always feasible to decompose the query into separate IR and DB parts). Informix uses user defined functions through its Datablades, with Excalibur providing its RetrievalWare software for text search.

IBM DB2 supports an external integration approach where the text extender, through external table function calls [39], contacts an external server and simulates a result table through thresholding and does not provide transactional semantics.

None of the theoretic nor implementation attempts ever mention support for query refinement. Several IR systems do support this but they are not integrated with a database system in any way. The most notable exception is INQUERY. When this system is layered on top of an RDBMS [166], it does provide for relevance feedback, but the RDBMS is used as a file system not a fully integrated (from the query processing angle) IR–DBMS.

The external integration approach is the most popular one but suffers from some shortcomings. The use of a threshold to convert an imprecise query into a set requires more work by the application and is the wrong abstraction. Also, no index can be exploited since there is no global optimization and an extensive search needs to be done.

Chapter 3

Data and Query Model

3.1 Introduction

This section describes our approach for an extended DBMS that supports similarity-based querying, ranking, rich data types, and query refinement to support content-based retrieval in addition to precise retrieval. Our approach is general since it supports any type of data and is not tied to a specific domain. The proposed system architecture follows the object-relational model and is extensible with user-defined types and functions. The set of types and functions defined by an application is collectively referred to as an Application Module Plug-in. In addition to the usual object-relational functionality (ADTs and UDFs), we allow applications to define (1) similarity functions, (2) functions that provide content extraction, and (3) functions that enable query refinement.[1] We present a data and matching model based on three major concepts:

- *Ranked sets* are an extension of relations with one extra numeric score attribute for each tuple. This score induces a total order or ranking in the relation.
- *Similarity-based predicates* that match objects based on their content. These predicates return a score interpreted as a similarity value in the range [0,1] where 1 means total similarity. Our model supports the mixing of similarity and precise conditions in the same query as well as using all the traditional relational operators in a suitably modified fashion.
- The *session*, or *iterative query* is a generalization of querying where a query is posed repeatedly with small modifications made by the DBMS using user input on the relevance of answers.

A complete retrieval model consists of (1) a data (object) model, (2) a (similarity-based) matching model, and (3) a query refinement model. To support our view of content-based retrieval, some changes are needed to relations, predicates, operators, and functions. These are described in the following sections.

[1]Functions that cooperate with a query optimizer (e.g., to provide cost estimates) are also considered but not discussed further here.

3.2 Data Model

The data model we follow is based on the object-relational model. The user creates new types using base types and adds functions to operate on these types. Some of the user specified functions are mandatory and address such things as input and output conversion, similarity, query refinement, etc. Other type-specific functions are allowed too, for example to crop an image.

Definition 3.2.1 (Type) *A type is a template for possible values and a set of functions, predicates and operators that operate on these values and define their behavior. Types can be simple ("built-in"), or composed based on arrays, structures and inheritance. The* domain *of a type is the set of all such values.* ■

Definition 3.2.2 (Object) *An object is one of all possible values in the domain of a type, this is also called an instance of its type: $object \in domain(type)$.* ■

In the rest of this paper, we use type and domain interchangeably for simplicity.

Example 3.2.1 (Geographic location) *An example data type for two dimensional geographic locations (see example 1.0.2) is the type* location(x: real, y: real)*, and (7,9) is an example object based on this type.* ■

Objects are values stored and managed in the database server and are logically arranged into rows called tuples. We extend the notion of a tuple with an extra *score* attribute[2] that is a number in the range $[0, 1]$.

Definition 3.2.3 (Tuple) *Given a set of n pairs $\langle a_i, D_i \rangle, i = 1, \ldots, n$ called a* schema *where a_i is a label called an* attribute *and D_i is a domain based on type T_i, and a pair $\langle score, [0, 1] \rangle$ that denotes a special attribute named* score *and its domain is a real number between 0 and 1, a* **tuple** *t is an element of the Cartesian product of $[0, 1]$ and $D_1, D_2, \ldots, D_n$: $t \in [0, 1] \times D_1 \times D_2 \times \cdots \times D_n$. The ith value (attribute) of tuple t is referred to by $t.a_i$ and has a domain of values of D_i; $t.score$ is the score of tuple t.* ■

Definition 3.2.4 (Ranked Set) *A ranked set RS is a set of tuples t with $t.score > 0$, and following definition 3.2.3 it is a subset of the Cartesian product of $(0, 1]$ and $D_1, D_2, \ldots, D_n$: $RS \subseteq (0, 1] \times D_1 \times D_2 \times \cdots \times D_n$. The jth tuple in a ranked set RS is denoted by $t_j(RS)$, its ith attribute as $t_j(RS).a_i$ and its content is an object of the domain (type) D_i. The attribute score induces a total order in the set. Assume there are m tuples, the ranked set with the tuples $t_1, t_2, \ldots t_m$ is ordered based on the score attribute as: $t_1.score \geq t_2.score \geq \ldots \geq t_m.score$. The rank of tuple t_j is j.* ■

[2]For practical purposes, tuples have yet one more attribute named *tid* which is a unique tuple identifier and its domain is the set of natural integers. We do not consider this attribute further.

Score	*Name*	*Age*	*Gpa*
0.8	Jim	20	3.8
0.6	Jane	22	4.0
0.2	Joe	19	2.5
...	...	...	...

Figure 3.1: Example Ranked Set

Thus, ranked sets contain only tuples with a score strictly larger than 0 and are logically (though not necessarily physically) always sorted on the score value. For base ranked sets (tables), this score is 1 for all tuples. In fuzzy or probabilistic databases, this score may be interpreted as the degree of membership or probability that a tuple is in a relation [118]. Here, we are not interested in representing uncertainty in the data itself, but in the similarity-based predicates, i.e., which tuples better fulfill a query.

Example 3.2.2 (Ranked Set) *Figure 3.1 shows an example ranked set over a table with three attributes: name, age, and gpa.* ■

3.3 Similarity Matching Model

A domain (type) supports a number of related predicates and operators that are applicable to that domain. An example set of operators which can be used whenever a domain has a total order (such as numeric values) is the set $\{<, \leq, =, \neq, \geq, >\}$. The conventional object-relational model is based on precise (also called crisp) retrieval semantics. We retain precise predicates and operators and provide alternate versions that extend such precise predicates to imprecise, score-based versions. The need to retain precise querying is natural since it conforms to the current practice in databases which we seek to complement, not replace. Precise predicates and operators return a score value in the (binary) set $\{0, 1\}$ for the traditional $\{false,\ true\}$ Boolean values, for similarity predicates and operators we generalize this as follows:

Definition 3.3.1 (Similarity Predicates and Operators) *A similarity-based version of each precise predicate and operator is defined. These similarity-based versions return a score value in the* range $[0, 1]$ *with 1 indicating total similarity and 0 indicating no similarity. Predicates can involve an attribute and query value(s) for a selection, or two attributes for a join predicate.* ■

For user-defined types, these are user supplied, while the system supplies inter-domain operators for conjunction, disjunction and negation. The similarity version is selected by prepending a "∼" to the name of the precise predicate or operator. The meaning of the similarity based operators (∼and, ∼or, ∼not) is not fixed to any particular interpretation, it can be based on probability theory, fuzzy logic, or some ad-hoc domain specific interpretation.

Example 3.3.1 (Precise vs. Similarity Predicate) *In example 1.0.2, while searching for high paying jobs, the applicant may use the greater than operator. The predicate salary* $> 80,000$ *returns* *false or true (0 or 1 here) depending on the salary listed in a job offer. Under an example similarity interpretation, salary* $\sim>$ 80,000 *has a score of 1 if salary* $>$ 80,000, *0 if salary* $<$ 50,000 *and score* $= \frac{salary-50{,}000}{30{,}000}$ *for* $50{,}000 \leq salary \leq 80{,}000$ *that includes lower paying jobs that might be of interest but with a lower score. (Note that score* $\in$ *[0,1]).* ■

While precise predicates operate on precise values, similarity-based predicates, due to their intrinsic imprecise nature, accept multiple query values where the precise version accepts only a single value.

Definition 3.3.2 (Multi-point Similarity Predicate) *Similarity predicates accept a set of values where the precise predicate version accepts a single value. This is termed a* multi-point query *as opposed to using a single query value which is a* single-point query *[169, 125]. The interpretation of evaluating a predicate with a set of values is chosen by the developer of the user-defined type.* ■

Example 3.3.2 (Multi-Point Similarity Predicate) *With geographic locations, a common desire is to find places similar or close to one or more places. Using example 3.2.1, location* $\sim=$ "$(7,9),(8,9)$" *may be interpreted as location close to* $(7,9)$ *or* $(8,9)$. ■

Given that similarity can be interpreted in different ways, we extend similarity predicates with an optional set of parameters that affect the behavior of the similarity predicate.

Definition 3.3.3 (Similarity predicate parameters) *Following a (single– or multi–point) similarity predicate, an optional set of configuration parameters can be specified enclosed in parenthesis. These parameters are passed "as is" to the function that implements the similarity computation, and remain constant during the execution of a query.* ■

Example 3.3.3 (Similarity predicate parameters) *Example 3.3.2 shows a similarity predicate based on Euclidean distance with respect to two query points. We want to give preference to points that relate to query points in a certain orientation. To achieve this goal, we specify weights for each dimension to emphasize the dimension's importance. The predicate is then: location* $\sim=$ "$(7,9),(8,9)$"$(0.8,0.2)$ *where alignment in the* x *dimension is more important (0.8) than alignment in the* y *dimension (0.2).*[3] ■

Conditional expressions are used to compute scores based on values contained in attributes and/or literals and are based on both, precise and similarity-based predicates and operators.

Definition 3.3.4 (Conditional Expressions) *Conditions, or conditional expressions are formed as conventional Boolean expressions. These expressions are constructed from atomic predicates on values and attributes that are defined for each type, these predicates can be precise or similarity based. Similarity based predicates are augmented with an optional weight parameter that indicates*

[3]A general rotation would require a rotation matrix, here we restricted orientation to the principal axes for demonstration purposes only.

the importance of the predicate to the query. We place this weight after the predicate enclosed in sqare brackets. The interpretation of this weight is up to a particulat implementation (see chapter 4). Expressions may be combined with precise Boolean conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$) operators, or similarity Boolean conjunction ($\sim \wedge$), disjunction ($\sim \vee$), and negation ($\sim \neg$) operators. Parentheses "( )" are used to nest expressions. ■

Example 3.3.4 (Conditional Expression) *An example of a conditional expression p (cf. example 1.0.2) is* $p = (salary > 80,000 \wedge (location \sim= \text{“}(7,9),(8,9)\text{”}[2.0] \ \sim \vee \ job\_position = \text{“}programmer\text{”}[1.0]))$ *where* $salary > 80,000$, *and* $job\_position = \text{“}programmer\text{”}$ *are precise predicates, and* $location \sim= ((7,9),(8,9))$ *is a similarity predicate, and* $\wedge$ *is a precise operator while* $\sim \vee$ *is a similarity* or *operator. Here,* $location \sim= \text{“}(7,9),(8,9)\text{”}[2.0]$ *has a weight of 2.0, and is more important than* $job\_position = \text{“}programmer\text{”}[1.0]$ *which has a weight of 1.0.* ■

We use $p(t)$ to denote the result of computing the score for a tuple t based on the conditional expression p: $t.score = p(t)$. Conditions may contain user defined precise operators that return values in $\{0, 1\}$ and similarity predicates that return values in the range $[0, 1]$. While the interpretation of similarity operators is provided by a system realization (cf. chapter 4), we must provide an interpretation for precise predicates that is robust in the presence of scores in the range $[0, 1]$:

Definition 3.3.5 (Precise Operator Semantics) *Precise operators follow the traditional Boolean interpretation, yet, since they manipulate scores, we must fix their interpretation in such a way that is consistent with Boolean semantics if only precise predicates are present, and still be compatible with the use of scores. We thus fix their semantics as follows:* $\wedge$ *uses the* min *and* $\vee$ *the* max *among the scores, and* $\neg$ *uses* $(1 - score)$, *this interpretation is consistent with traditional Boolean semantics. Interpretation of the similarity versions is left to a system realization (cf. chapter 4). The precedence among these operators in decreasing order is:* $\neg, \sim \neg, \wedge, \sim \wedge, \vee, \sim \vee$. ■

Note that although this appears to be a fuzzy semantics of operators, it is really a natural extension of precise operators to handle scores, i.e, for a condition $p = (a = 1 \wedge b \sim> 0)$, if $b \sim> 0$ has a score of 0.8, then $a = 1$ may be either 0 or 1. The overall score should naturally reflect that if $a = 1$ is true, then the score should be that of $b \sim> 0$ which is 0.8.

Example 3.3.5 (Conditional Expression Evaluation) *Following example 3.3.4, if* $p = (salary > 80,000 \wedge (location \sim= ((7,9),(8,9)) \ \sim \vee \ job\_position = \text{“}programmer\text{”}))$ *and a tuple* t_1 =(score, 70,000, (0,0), "programmer"), *we compute* $p(t_1)$ *as a combination of the scores of individual predicates:* $70,000 > 80,000$ *is 0,* $(0,0) \sim= ((7,9),(8,9))$ *is 0.7 under some interpretation of* $\sim=$ *for location, and "programmer" = "programmer" is 1 since they are equal. To combine the scores 0.7 and 1 with the* $\sim \vee$ *operator, an interpretation must be provided (cf. chapter 4), here we can choose a probabilistic interpretation [115], thus the score for* $(0,0) \sim= ((7,9),(8,9)) \ \sim \vee$ *"programmer" = "programmer" is* $(0.7 + 1.0) - (0.7 \times 1.0) = 0.7$. *To combine this with the score for* $70,000 > 80,000$ *which is 0 with the* $\wedge$ *operator, we use* $\min(0, 0.7) = 0$ *(definition 3.3.5)* *and the final result is* $p(t_1) = 0$*. For another tuple* t_2 =(score, 90,000, (0,0), "manager")*, the final score is* $p(t_2) = \min(1, (0.7 + 0) - (0.7 \times 0)) = 0.7$ *under the above assumptions. In this example, we ignore the optional weights since their interpretation depends on the actual model for similarity operators, this is discussed in chapter 4.* ∎

We now present an algebra that uses ranked sets and conditional expressions that involve precise and similarity predicates and operators.

3.3.1 Ranked Set Algebra

We develop a ranked set algebra that extends the relational algebra to support our model and present how the operators change. We present interpretations for a *top* operator and the basic relational algebra operators: project (σ), select (π), join ($\bowtie$), cross product ($\times$), union ($\cup$), and difference ($-$). Two of these operators (σ, $\bowtie$), which include a condition to determine which tuples are output and their ranking, have two versions. One that uses the original ranking of the input ranked sets and combines it with the outcome of evaluating the condition, and a second version (denoted by $\widehat{\sigma}$, and $\widehat{\bowtie}$) that replaces the score of the input relations regardless of the original score or ranking. The purpose of these alternate operators that ignore original ranking is for query optimization and will become clear later.

For each ranked set RS we also maintain a set of attributes U denoted by RS^U where the set U includes those attributes that may affect the ranking of the output. This set is kept so that columns that affect the ranking are not removed by projection but instead merely hidden away from the user, the purpose being for query refinement algorithms to access and re-use the values of attributes that affect the ranking.

Definition 3.3.6 (Ranking Attributes) *For a condition expression* p*, let* $ranking\_attributes(p)$ *denote the set of attributes in* p *that affect the ranking of a tuple.* ∎

Example 3.3.6 (Ranking Attributes) *Following example 3.3.5, for the expression* $p = (salary > 80,000 \wedge (location \sim= (7,9) \ \sim\vee \ job\_position = \text{"}programmer\text{"}))$ *the ranking attributes are:* $ranking\_attributes(p) = \{location,\ job\_position\}$*. The location attribute is trivially included since it participates in a similarity predicate. The attribute job_position is also included despite* $job\_position = \text{"}programmer\text{"}$ *being a precise expression, since the* $\sim\vee$ *(similarity or) uses its result which can affect the ranking depending on the model chosen for* $\sim\vee$*. The reason to exclude salary is that the precise predicate salary* $> 80,000$*, in conjunction with the precise* $\wedge$ *(*and*) operator, has the effect of either preserving the score computed by other subexpressions, or setting it to 0 thus indicating a tuple with that score to be removed and does not otherwise affect any ranking in a ranked set. Another example is* $q = (location \sim= (7,9) \vee \ job\_position = \text{"}programmer\text{"})$*, where* $ranking\_attributes(q) = \{location,\ job\_position\}$ *since location participates in a similarity predicate and the result of* $job\_position = \text{"}programmer\text{"}$ *may increase the score or leave it alone which may affect ranking.* ∎

We define the following notation for the discussion: $x \prec y$ denotes that tuple x is ranked better than tuple y ($x.score > y.score$), $x \preceq y$ denotes that x and y are equally ranked or $x \prec y$ ($x.score \geq y.score$), conversely, $\succ$, and $\succeq$ denote the symmetric relationships. We use $x \equiv y$ to denote tuples that have equal attributes except for the *score* ($\forall a \in schema(\{x,\ y\}) - \{score\},\ x.a = y.a$) (cf. definition 3.2.3). For example, if $x = (0.8, \text{“}Joe\text{”}, 31, \text{“}SQL\text{”}, \text{“}NY\text{”})$ and $y = (0.6, \text{“}Al\text{”}, 25, \text{“}C++\text{”}, \text{“}NJ\text{”})$, then $x \prec y$.

3.3.1.1 Top

We present two variations of a *top* operator. One version of top denoted by $top(RS^U)$ returns the top ranked tuple in RS^U: $top(RS^U) = t \mid (t \in RS^U) \wedge (\forall x \in RS^U - \{t\},\ t \preceq x)$. Since RS^U is ranked, this is simply the first tuple. The second variation returns a new ranked set with the n top ranked tuples (denoted by the subscript n): $top_n(RS^U) = \{t \mid (t = top(RS^U) \vee t \in top_{n-1}(RS^U - \{top(RS^U)\})) \wedge n \geq 1 \wedge RS^U \neq \emptyset\}$. The set of attributes U is unchanged. We assume there is a special value for n denoted by *all* which makes $n = |RS^U|$ and returns the whole ranked set RS^U. Note that trivially $RS^U = top_{all}(RS^U)$.

3.3.1.2 Projection

The projection operator π reduces the number of columns of a relation or ranked set. If S is a subset of attributes of RS^U, then $\pi_S(RS^U)$ has the columns in $(S \cup U) \cap schema(RS^U)$. The ranked set, after projecting on the attributes in $(S \cup U)$, obeys the original ranking, i.e., if tuple x is ranked above tuple y after the projection, x was also ranked above y before the projection: $x \in RS^U \wedge y \in RS^U \wedge x \preceq y \wedge x_1 = \pi_{S \cup U}(x) \in \pi_S(RS^U) \wedge y_1 = \pi_{S \cup U}(y) \in \pi_S(RS^U) \Rightarrow (x_1 \preceq y_1 \vee x_1 \equiv y_1)$. The attributes in the set U and not in the list of attributes to project $(U - S)$ are marked as hidden attributes and are present in the result, but not returned to the user. Projection may result in duplicate tuples ($x \equiv y$) which are removed by choosing the highest scoring tuple among the duplicates. For projection, the set of attributes U is unchanged and the projection is computed as:

$$\begin{aligned} \pi_S(RS^U) = \{t \mid\ & t_1 \in RS^U \wedge t.score = t_1.score \wedge (\forall a \in schema(RS^U) \cap (S \cup U),\ t.a = t_1.a) \wedge \\ & (\nexists t_2 \in RS^U \mid \forall a \in schema(RS^U) \cap (S \cup U),\ t_2.a = t.a \wedge t_2 \preceq t)\} \end{aligned}$$

The first part of the formula keeps those attributes of t that are requested in S or are in U, while the second line of the formula removes duplicates.

3.3.1.3 Selection

The *select* operator σ removes and/or re-ranks tuples from a ranked set using a *selection condition*. A selection condition p may be either precise, (i.e., $salary = 50,000$), similarity-based (i.e., $location \sim= (7,9)$), or mixed (i.e., $salary = 50,000 \sim \wedge\ location \sim= (7,9)$). Since selection uses a condition, there are two versions of the operator, $\sigma_p(RS^U)$ uses the original ranking and scores

from RS^U and combines them with the evaluation of p, and $\widehat{\sigma}_p(RS^U)$ which ignores the original ranking and replaces the score with the result of evaluating p:
$\sigma_p(RS^V) = \{t \mid t_1 \in RS^V \wedge t \equiv t_1 \wedge t.score = \min(t_1.score, p(t_1))\}$
$\widehat{\sigma}_p(RS^V) = \{t \mid t_1 \in RS^V \wedge t \equiv t_1 \wedge t.score = p(t_1)\}$
For both operators, the attribute set U is augmented with those attributes of the condition that affect the ranking: $U = V \ \cup \ ranking\_attributes(p)$. The σ operator uses *min* to incorporate the original ranking with p, this acts as a precise conjunction and is consistent with the conventional relational algebra equivalence $\sigma_p(\sigma_q(RS^U)) = \sigma_{p\wedge q}(RS^U)$. If p is precise, then the ranking is not altered, only some tuples are returned. If p is similarity-based or mixed, then re-ranking may occur.

Example 3.3.7 (Image Retrieval) *An example selection for our multimedia example that selects wildlife images and ranks them on image features is:*
$\sigma_{(color\sim=c_1 \ \sim\wedge \ texture\sim=t_1)\wedge category=\text{“}wildlife\text{”}}(image\_table)$. *This is equivalent to*
$\sigma_{color\sim=c_1 \ \sim\wedge \ texture\sim=t_1}(\sigma_{category=\text{“}wildlife\text{”}}(image\_table))$. *The $\widehat{\sigma}$ operator can be used to push a similarity condition as in:* $\widehat{\sigma}_{color\sim=c_1 \ \sim\wedge \ texture\sim=t_1}(\sigma_{texture\sim=t_1\wedge category=\text{“}wildlife\text{”}}(image\_table))$. *Subexpressions that affect ranking are retained in the $\widehat{\sigma}$ operator to correctly compute scores.* ■

3.3.1.4 Join and Cross Product

The join operator $\bowtie$ forms a new ranked set based on two input ranked sets and a join condition p which can be precise, approximate or mixed. The *cross product* is a special case of join with a precise join condition that always evaluates to true ($\bowtie_{true}$). In join, tuples from the two inputs are combined into a new tuple by combining all the attributes of the input ranked sets. This tuple is then subjected to the join condition and the score appropriately updated:

$$\begin{aligned} RS_1^V \bowtie_p RS_2^W = \{t \mid\ & t_1 \in RS_1^V \wedge t_2 \in RS_2^W \wedge schema(t) = schema(t_1) \cup schema(t_2) \wedge \\ & (\forall a \in schema(t_1) - \{score\},\ t.a = t_1.a) \wedge t.score = \min(t_1.score, t_2.score, p(t)) \wedge \\ & (\forall b \in schema(t_2) - \{score\},\ t.b = t_2.b)\} \end{aligned}$$

The ranking attribute set U is computed as: $U = V \cup W \cup ranking\_attributes(p)$. As in selection, the ranking of the inputs is preserved by combining it with the outcome of the join condition as a precise conjunction (*min*).

The join operator, which supports a join condition, has an alternate version that replaces the original score with that computed by the condition:

$$\begin{aligned} RS_1^V \widehat{\bowtie}_p RS_2^W = \{t \mid\ & t_1 \in RS_1^V \wedge t_2 \in RS_2^W \wedge schema(t) = schema(t_1) \cup schema(t_2) \wedge t.score = p(t) \wedge \\ & (\forall a \in schema(t_1) - \{score\},\ t.a = t_1.a) \wedge (\forall b \in schema(t_2) - \{score\},\ t.b = t_2.b)\} \end{aligned}$$

As above, the ranking attribute set U is computes as: $U = V \ \cup \ W \ \cup \ ranking\_attributes(p)$.

This interpretation in general requires computing the entire answer and then sorting it on the score. In our context, the primary purpose of this operator is to push selections down, and utilizing their results. While this operator re-assigns the score, we assume that inputs are already ranked on subexpressions of the join condition. Section 3.3.2 presents cases where this is useful.

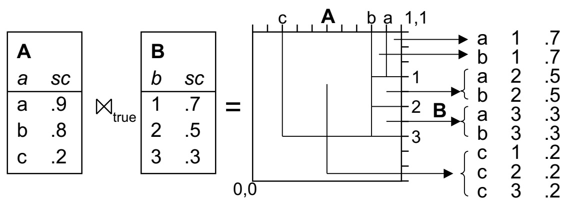


Figure 3.2: Ranked Set Join

Example 3.3.8 (Job Applications) *Our job application example (example 1.0.2) has the base ranked sets:* applicant(name, age, resume, home_location, desired_salary) *and* jobs(description, location, salary, position)*. The following query matches applicants to positions based on skills and salary:*
$applicant \bowtie_{resume \sim= description\ \sim\wedge\ desired\_salary \sim= salary}\ jobs.$ ■

Example 3.3.9 (Join Results) *Figure 3.2 shows the results we expect from a join without a condition between tables A and B. The output is a ranked set.* ■

3.3.1.5 Union

The union ($\cup$) of two ranked sets produces a ranked set with tuples from both input ranked sets. A tuple appears in the answer RS^U because it appears in either RS_1^V, or RS_2^W, or in both. If a tuple appears in both (a duplicate), the tuple with the higher score is returned and any duplicates discarded. Tuples that appear in RS^U due to RS_1^V follow the original ranking of RS_1^V while tuples is RS^U due to RS_2^W follow the original ranking of RS_2^W:

$$RS_1^V \cup RS_2^W = \{t \mid (t \in RS_1^V \wedge t \notin RS_2^W) \vee (t \in RS_2^W \wedge t \notin RS_1^V) \vee (t_1 \in RS_1^V \wedge t_2 \in RS_2^W \wedge t \equiv t_1 \equiv t_2 \wedge t.score = \max(t_1.score,\ t_2.score))\}$$

The attribute set U combines V and W: $U = V \cup W$.

3.3.1.6 Difference

The difference operator ($-$) subtracts RS_2^W from RS_1^V by adjusting the scores of tuples. The score for a tuple t is the difference of the scores for t in RS_1^V and RS_2^W: $RS_1^V - RS_2^W = \{t \mid (t \in RS_1^V \wedge t \notin RS_2^W) \vee (t_1 \in RS_1^V \wedge t_2 \in RS_2^W \wedge t \equiv t_1 \equiv t_2 \wedge (t.score = t_1.score - t_2.score) > 0)\}$. The attribute set U is computed as: $U = V \cup W$.

3.3.2 Equivalences in Ranked Set Algebra

In this section we present some equivalences and containments over the new algebra for ranked sets we presented above. Relevant transformations are presented whose goal is to enable rewriting of algebra expressions to equivalent algebra expressions. These results can be used to lower the cost of query processing, for example, by heuristically pushing down selections through joins. Selecting among different equivalent expressions (query optimization), although an important problem, is outside the scope of this paper due to space restrictions. We denote conditions as follows: $\overline{p}$ has only precise, $\widetilde{p}$ has only similarity-based, and $\widetilde{\overline{p}}$ has both, similarity-based and precise predicates and/or operators; p denotes any of the above three variations.

Projection. Projection is involved with many operators, specially to reduce the size of tuples and thus query processing time. A useful rule that only involves projection is: $\pi_A(\pi_B(RS)) = \pi_B(\pi_A(RS)) = \pi_{A \cap B}(RS)$.

Selection. The select operator follows many of the conventional equivalences for precise only conditions, e.g., $\sigma_p(\sigma_q(RS)) = \sigma_q(\sigma_p(RS)) = \sigma_{p \wedge q}(RS)$, and $\sigma_{p \vee q}(RS) = \sigma_p(RS) \cup \sigma_q(RS)$, are valid for both precise and similarity-based conditions. Similarity-based conditions however invalidate some equivalences and offer opportunities for others. Given the interpretations of similarity-based operators, the $\widehat{\sigma}$ operator must be used to push selections: $\sigma_{p\ \sim\wedge\ q}(RS) = \widehat{\sigma}_{p\ \sim\wedge\ q}(\sigma_p(RS)) = \widehat{\sigma}_{p\ \sim\wedge\ q}(\sigma_q(RS)) \quad (E1)$, and $\sigma_{p\ \sim\vee\ q}(RS) = \widehat{\sigma}_{p\ \sim\vee\ q}(\sigma_p(RS)) = \widehat{\sigma}_{p\ \sim\vee\ q}(\sigma_q(RS)) \quad (E2)$. Pushed subexpressions partially pre-rank input to the $\widehat{\sigma}$ operator.

Join. The join operator is traditionally one of the most costly and common operators, and thus one for which equivalences are quite important. Pushing selections into a join is a common equivalence which holds under our model as long as precise conjunction is used: $\sigma_{p \wedge q}(RS_1 \bowtie_o RS_2) = \sigma_p(RS_1) \bowtie_o \sigma_q(RS_2)$ if p involves RS_1 only, and q RS_2 only. Similarly, join is associative for precise conditions, but may not be for conditions involving similarity: $(RS_1 \bowtie_{\overline{p}} RS_2) \bowtie_{\overline{q}} RS_3 = RS_1 \bowtie_{\overline{p}} (RS_2 \bowtie_{\overline{q}} RS_3)$. If the operators involved are not precise, then the $\widehat{\bowtie}$ operator must be used; if p involves only RS_1 and q involves RS_2 and o is a join condition: $\sigma_{\widetilde{\overline{p}}\ \sim\wedge\ \widetilde{\overline{q}}\ \sim\wedge\ \widetilde{\overline{o}}}(RS_1 \bowtie RS_2) = \sigma_{\widetilde{\overline{p}}}(RS_1) \widehat{\bowtie}_{(\widetilde{\overline{p}}\ \sim\wedge\ \widetilde{\overline{q}}\ \sim\wedge\ \widetilde{\overline{o}})} \sigma_{\widetilde{\overline{q}}}(RS_s) \neq \sigma_{\widetilde{\overline{p}}\ \sim\wedge\ \widetilde{\overline{q}}}(RS_1 \bowtie_{\widetilde{\overline{o}}} RS_2) = \sigma_{\widetilde{\overline{p}}}(RS_1) \widehat{\bowtie}_{(\widetilde{\overline{p}}\ \sim\wedge\ \widetilde{\overline{q}}) \wedge \widetilde{\overline{o}}} \sigma_{\widetilde{\overline{q}}}(RS_s)$. Notice that $\sigma_{\widetilde{\overline{p}}}$ and $\sigma_{\widetilde{\overline{q}}}$ are pre-computed and ranked for the join, and that conditions with similarity operators may prevent clean separation and migration of a join subexpression from a selection to a join. A rather trivial equivalence is $RS = (RS\ as\ X) \bowtie_{\forall a \in schema(RS) - \{score\}, X.a = Y.a} (RS\ as\ Y)$ which says that a ranked set is equivalent to joining with itself (after properly adjusting the schema). This seems of limited applicability, but is useful in other equivalences (combining with $(E1)$ and $(E2)$ above): $\sigma_{p\ \sim\wedge\ q}(RS) = (\sigma_p(RS)\ as\ X) \widehat{\bowtie}_{(p\ \sim\wedge\ q) \wedge (\forall a \in schema(RS) - \{score, attributes(p,q)\}, X.a = Y.a)} (\sigma_q(RS)\ as\ Y)$, and $\sigma_{p\ \sim\vee\ q}(RS) = (\sigma_p(RS)\ as\ X) \widehat{\bowtie}_{(p\ \sim\vee\ q) \wedge (\forall a \in schema(RS) - \{score, attributes(p,q)\}, X.a = Y.a)} (\sigma_q(RS)\ as\ Y)$ for conjunction and disjunction respectively. These equivalences embody Fagins work on rank merging [46] where different sources (here σ_p, σ_q) produce different rankings for the same objects (tuples) which are then merged.[4] Incremental algorithms for fuzzy and probabilistic interpretations of the similarity operators appear in [115] and are easily extended to an ad-hoc weighted summation interpretation.

[4]To exactly model [46], the top_n operator also discussed in this section is applied to this result to limit the answers.

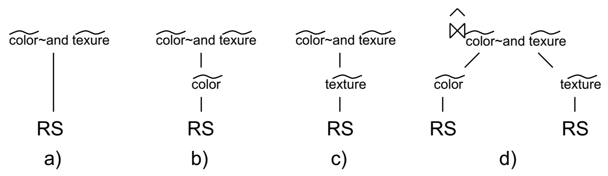


Figure 3.3: Equivalent Expressions for an Image Retrieval Condition

Top-n. The top_n operator interacts with other operators to limit the output of tuples. As long as there are no duplicates, project and top commute as: $\pi_a(top_n(RS)) = top_n(\pi_A(RS))$, if duplicates are possible in the projection, then $\pi_a(top_n(RS)) \subseteq top_n(\pi_A(RS))$. *Top* also affects selection in the following way: $\sigma_p(top_n(RS)) \subseteq top_n(\sigma_p(RS))$. Top distributes over union as: $top_n(RS_1 \cup RS_2) \subseteq top_n(RS_1) \cup top_n(RS_2)$, which by adding another *top* operator becomes $top_n(RS_1 \cup RS_2) = top_n(top_n(RS_1) \cup top_n(RS_2))$. A similar transformation for the difference operator is: $top_n(RS_1 - RS_2) = top_n(top_{n+|RS_2|}(RS_1) - RS_2)$.

Example 3.3.10 (Alternative Image Retrieval Conditions) *Using our multimedia example (example 1.0.1), $\widetilde{color} = (color \sim= (c1, c2))$, and $\widetilde{texture} = (texture \sim= (t1))$ rank images on color and texture similarity. A query can be done as a selection $\sigma_{\widetilde{color} \sim\wedge \widetilde{texture}}(RS)$ that can be supported by a sequential scan. It is also possible to push either the color or texture subexpressions into another selection, perhaps to take advantage of available indices: $\widehat{\sigma}_{\widetilde{color} \sim\wedge \widetilde{texture}}(\sigma_{\widetilde{color}}(RS))$ $= \widehat{\sigma}_{\widetilde{color} \sim\wedge \widetilde{texture}}(\sigma_{\widetilde{texture}}(RS))$. If there are indices on both features, we may independently use the indices for $\widetilde{color}$ and $\widetilde{texture}$ followed by merging: $(\sigma_{\widetilde{color}}(RS)\ as\ X) \widehat{\bowtie}_{(\widetilde{color} \sim\wedge \widetilde{texture}) \wedge X.id=Y.id}(\sigma_{\widetilde{texture}}(RS)\ as\ Y)$ Figure 3.3 shows these expressions graphically.* ■

This list of equivalences and containments is by no means complete. We focussed on some rules that affect our proposed model, while the usual rules for precise-only queries still apply.

Chapter 4

Interpretation of Similarity Conditions

4.1 Introduction

The similarity matching model and algebra described in the previous chapter are independent of any particular interpretation of the predicates used for ranking. This chapter presents three possible instances of many possible interpretations based on weighted summation, fuzzy and probabilistic models. Thorough explorations of aggregation models can be found in [41, 47, 46, 42], here we limit ourselves to basic interpretations and extend them with weighting capabilities to take advantage of weights specified for similarity predicates. We extend the linear weighting approach from [47] while maintaining its basic properties.

Query conditions can be of two types, crisp conditions which are exact matches and similarity expressions which serve to rank the results, we also call them *ranking expressions*. The crisp conditions follow the traditional Boolean model of true and false which can be represented by a score of 1 and 0 respectively.

A ranking expression has *intra-domain* predicates and composite *inter-domain* expressions. In this chapter, we focus on the interpretation of inter-domain, or inter-predicate expressions, that is, similarity conjunctions, disjunctions and negations.

We discuss three possible interpretations of similarity score aggregation: a weighted summation model, a fuzzy logic based model and a probabilistic model.

4.2 Weighted Summation Interpretation

One simple interpretation for combining similarity scores for a tuple is based on *weighted summation*. In this model, we aggregate similarity scores for conjunctions and disjunctions using weighted L_p metrics:

Definition 4.2.1 (L_p metric) *Let there be a set of weights $w_1, w_2, ...w_n$, and a corresponding set of similarity scores $s_1, s_2, ...s_n, s_i \in [0, 1]$. An L_p metric is then defined as:*

$$L_p = \sqrt[p]{\sum_{i=1}^{n} w_i \times s_i^p}$$

The familiar Euclidean distance is the L_2 metric. ■

Based on this definition, we define the similarity operators as follows:

Definition 4.2.2 (Similarity operators) *We use the L_p metric to define our similarity operators:*

- *$\sim$ and we use an L_p metric with a value of $p = 1$, this results in a linear combination of weights*
- *$\sim$ or we use an L_p metric with a value of $p = 2$, this simulates a disjunction among the similarity scores (see figure 5.3c)*
- *$\sim$ not: $1 - s$* ■

Weights are initially attached only to predicates, to use L_p metrics in this model, we need to convert the weights assigned to each predicate to weights for each conjunct in a conjunction and each disjunct in a disjunction.

Definition 4.2.3 (Weight computation) *Let $e_1, e_2, ...e_n$ be similarity expressions with weights w_i for each expression. An expression can be a similarity predicate, a conjunction or a disjunction. Let $e_1 \star e_2 \star ...e_n$ be either a conjunction or disjunction of the expressions. For this model, we enforce a restriction that all the weights for a conjunction or disjunction add up to one, that is $w'_i = \frac{w_i}{\sum_{i=1}^{n} w_i}$. We also assign an "overall" weight for the conjunction or disjunction by adding all the original weights: $\sum_{i=1}^{n} w_i$. This becomes the weight for the expression and is used as the expression's weight if this expression is part of another expression.* ■

Example 4.2.1 (Weight computation) *Let the ranking condition for a query be:*

$$(p_1[1] \sim and\ p_2[5] \sim and\ p_3[2]) \sim or\ (p_4[3]) \sim or\ (p_5[3] \sim and\ p_6[1])$$

There are three disjuncts overall: $(p_1[1] \sim and\ p_2[5] \sim and\ p_3[2])$, $p_4[3]$, and $(p_5[3] \sim and\ p_6[1])$. The first disjunct is a conjunction of three predicates, with p_2 being more important than p_1 and p_3. We convert this conjunction to: $(p_1[\frac{1}{8}] \sim and\ p_2[\frac{5}{8}] \sim and\ p_3[\frac{2}{8}])$, and an overall weight of $1 + 5 + 2 = 8$. The second and third disjuncts are also converted and result in:

	overall weight
$p_1[0.125] \sim and\ p_2[0.625] \sim and\ p_3[0.25]$	*0.533*
$p_4[1]$	*0.2*
$p_5[0.75] \sim and\ p_6[0.25]$	*0.2667*

■

Example 4.2.2 (Tuple score computation) *We use the sample ranking condition of example 4.2.1. Assume for a given tuple t the similarity scores for each similarity predicate are:* $p_1 = 0.8, p_2 = 0.7, p_3 = 0.6, p_4 = 0.5, p_5 = 0.9,$ *and* $p_6 = 1.0$. *Using the weights computed in example 4.2.1 we compute the overall similarity score for the tuple as follows:*
$(0.533 \times (0.125 \times 0.8 + 0.625 \times 0.7 + 0.25 \times 0.6)^{-2} + 0.2 \times (1 \times 0.5)^{-2} + 0.2667 \times (0.75 \times 0.9 + 0.25 \times 1.0)^{-2})^{-1/2} = 0.668$

The overall tuple score is then 0.668. ■

We combine weighted summation with crisp predicates in the same way but use min and max as described in chapter 3; crisp predicates always have a score of 0 or 1, and there are no weights involved.

Example 4.2.3 (Crisp and similarity tuple evaluation) *Assume the following condition in a query, where c_i stands for a crisp predicate while p_i stands for a similarity predicate:*

$$p_1[1] \sim \textit{and } p_2[3] \textit{ and } c_1 \textit{ or } p_3[1] \textit{ and } c_2$$

If s_{p_i} and s_{c_i} denote the scores of p_i and c_i respectively, the overall score for a tuple is computed as: $\max\left[\min\left\{(0.25 \times s_{p_1} + 0.75 \times s_{p_2}), s_{c_i}\right\}, \min\left\{1.0 \times s_{p_3}, s_{c_2}\right\}\right]$ ■

4.3 Fuzzy Interpretation

Fuzzy logic [170] has been successfully used for many applications. Under this interpretation, we interpret the similarity score resulting from matching a tuple with a similarity predicate as the *degree of membership* of the tuple to a *fuzzy* set formed by the similarity predicate. We then use fuzzy logic to combine the scores of multiple predicates into an overall score for the tuple. We aggregate similarity scores for conjunctions and disjunctions using weighted variations of *min* and *max*. We first discuss how we perform the matching and then how we derive the weights from the base predicates.

Definition 4.3.1 (Similarity operators) *Let there be a set of weights $w_1, w_2, ...w_n$, and a corresponding set of similarity scores $s_1, s_2, ...s_n, s_i \in [0,1]$ for a conjunction, disjunction or negation. We define the operators as:*

- $\sim$ *and:* $\min_{i=1}^{n} s_i^{\frac{1}{w_i}}$
- $\sim$ *or:* $\max_{i=1}^{n} s_i^{\frac{1}{w_i}}$
- $\sim$ *not:* $1 - s^{\frac{1}{w}}$

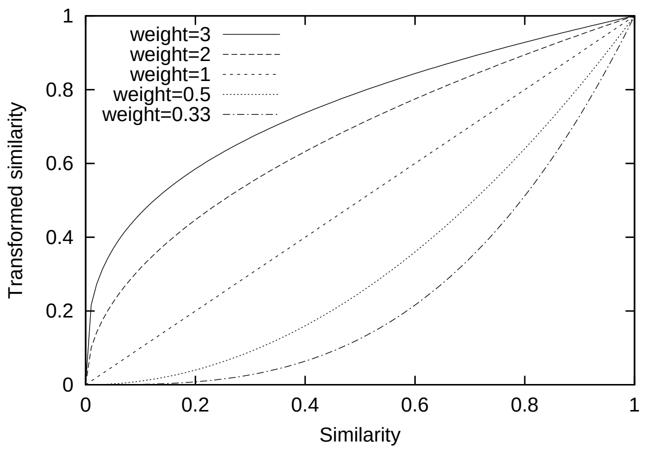


Figure 4.1: Various Sample Similarity Weightings

The motivation for using an exponent of $\frac{1}{weight}$ *for similarity scores is to ensure a smooth mapping for scores from* $[0,1] \rightarrow [0,1]$*, and to preserve the meaning that higher weights boost similarity scores more than lower weights. Figure 4.1 shows graphically how the weighting affects scores.* ■

Weights are initially attached only to predicates, to use this approach, we must convert the weights assigned to each predicate to weights for each conjunct in a conjunction and each disjunct in a disjunction.

Definition 4.3.2 (Weight computation) *Let* $e_1, e_2, ...e_n$ *be similarity expressions with weights* w_i *for each expression. An expression can be a similarity predicate, a conjunction or a disjunction. Let* $e_1 \star e_2 \star ...e_n$ *be either a conjunction or disjunction of all the expressions. We enforce a restriction that all the weights for a conjunction or disjunction* average *one, that is* $w_i' = \frac{w_i \times n}{\sum_{i=1}^{n} w_i}$. *The motivation for weights to average 1 is that this represents the identity, i.e., no change in the similarity score. We also assign an "overall" weight for the conjunction or disjunction by adding all the original weights:* $\sum_{i=1}^{n} w_i$*. This becomes the weight for the expression and is used in the same manner if this expression is part of another expression.* ■

Example 4.3.1 (Weight computation) *Let the ranking condition for a query be:*

$$(p_1[1] \sim \textit{and } p_2[5] \sim \textit{and } p_3[2]) \sim \textit{or } (p_4[3]) \sim \textit{or } (p_5[3] \sim \textit{and } p_6[1])$$

There are three disjuncts overall: $(p_1[1] \sim \textit{and } p_2[5] \sim \textit{and } p_3[2])$, $p_4[3]$, *and* $(p_5[3] \sim \textit{and } p_6[1])$*. The first disjunct is a conjunction of three predicates, with* p_2 *being more important than* p_1 *and* p_3*. We convert this conjunction to:* $(p_1[\frac{1\times3}{8}] \sim \textit{and } p_2[\frac{5\times3}{8}] \sim \textit{and } p_3[\frac{2\times3}{8}])$*, and an overall weight of* $1+5+2=8$*. The second and third disjuncts are also converted and result in:*

	overall weight
$p_1[0.375] \sim$ *and* $p_2[1.875] \sim$ *and* $p_3[0.75]$	*1.6*
$p_4[1]$	*0.6*
$p_5[1.5] \sim$ *and* $p_6[0.5]$	*0.8*

■

Example 4.3.2 (Tuple score computation) *We use the sample ranking condition of example 4.3.1. Assume for a given tuple t the similarity scores for each similarity predicate are: $p_1 = 0.8, p_2 = 0.7, p_3 = 0.6, p_4 = 0.5, p_5 = 0.9$, and $p_6 = 1.0$. Using the weights computed in example 4.3.1 we compute the overall similarity score for the tuple as follows:*
$\max(\min(0.8^{0.375}, 0.7^{1.875}, 0.6^{0.75})^{1.6}, 0.5^{0.6}, \min(0.9^{1.5}, 1.0^{0.5})^{0.8}) = 0.8812$

The overall tuple score is then 0.8812. ■

We combine the fuzzy interpretation with crisp predicates in the same way as described for the weighted summation model and use min and max as described in chapter 3; crisp predicates always have a score of 0 or 1, and there are no weights involved.

Example 4.3.3 (Crisp and similarity tuple evaluation) *Assume the following condition in a query, where c_i stands for a crisp predicate while p_i stands for a similarity predicate:*

$$p_1[1] \sim \textit{and } p_2[3] \textit{ and } c_1 \textit{ or } p_3[1] \textit{ and } c_2$$

If s_{p_i} and s_{c_i} denote the scores of p_i and c_i respectively, the overall score for a tuple is computed as:

$\max\left[\min\left\{\min(s_{p_1}^{0.5}, s_{p_2}^{1.5}), s_{c_i}\right\}, \min\left\{s_{p_3}^{1.0}, s_{c_2}\right\}\right]$ ■

4.4 Probabilistic Interpretation

Probability theory has a long and fruitful history which makes it a good candidate for similarity retrieval. Under this interpretation, we interpret similarity predicate scores as the probability that a given attribute value matches the query value(s). This interpretation is natural since similarity scores and probabilities are always in the range [0,1] and there is a positive association between them – higher values indicate better matches. When calculating or combining probabilities, one of the most complex issues is that of dependence or independence among different elements. In our scenario, this amounts to determining when two similarity predicates over a tuple are or not dependent. Since similarity predicates are user defined, it is not possible in general to determine whether they are or not dependent. Therefore we assume that all the similarity predicates involved in a query are independent. This also reduces the complexity of the calculations. In this model, we aggregate similarity scores for conjunctions and disjunctions using weighted variations of the common probabilistic conjunction, disjunction and negation formulas. We first discuss how we perform the matching and then how we derive the weights from the base predicates.

Definition 4.4.1 (Similarity operators) *Let there be a set of weights $w_1, w_2, ...w_n$, and a corresponding set of similarity scores $s_1, s_2, ...s_n, s_i \in [0,1]$ for a conjunction, disjunction or negation. We define the operators as:*

- $\sim$ *and:* $\prod_{i=1}^{n} s_i^{\frac{1}{w_i}}$
- $\sim$ *or:* $\sum\limits_{1 \le i \le n} s_i^{\frac{1}{w_i}} - \sum\limits_{1 \le i < j \le n} (s_i^{\frac{1}{w_i}} \times s_j^{\frac{1}{w_j}}) + \sum\limits_{1 \le i < j < k \le n} (s_i^{\frac{1}{w_i}} \times s_j^{\frac{1}{w_j}} \times s_k^{\frac{1}{w_k}}) - ... + (-1)^{n-1} \prod\limits_{i=1}^{n} s_i^{\frac{1}{w_i}}$, *this is the well known mutual inclusion–exclusion formula [126]*
- $\sim$ *not:* $1 - s^{\frac{1}{w}}$

Note that the inclusion–exclusion formula used for disjunction is of limited practical [126] use as its complexity grows too rapidly as the number of predicates increases.

The motivation for using an exponent of $\frac{1}{weight}$ for similarity scores is to ensure a smooth mapping for scores from $[0,1] \rightarrow [0,1]$, and to preserve the meaning that higher weights boost similarity scores more than lower weights. Figure 4.1 shows graphically how the weighting affects scores. ■

We use the same approach used in the fuzzy model (see section 4.3) to convert the weights attached to similarity predicates for use with our probabilistic model. We also combine the probabilistic interpretation of similarity expressions with crisp predicates in the same way as the fuzzy model.

4.5 Evaluation

We conducted extensive experiments of varied datasets to measure the performance of the retrieval models and query processing algorithms developed. This section presents the results of our experiments. First we briefly describe the parameters used to measure retrieval performance followed by a description of the data sets. Finally, we present the results along with our observations.

4.5.1 Methodology

Text retrieval systems typically use the following two metrics to measure the retrieval performance: *precision* and *recall* [137, 20]. Note that these metrics measure the retrieval performance as opposed to execution performance (retrieval speed).

Precision and recall are based on the notion that for each query, the collection can be partitioned into two subsets of documents. One subset is the set of relevant documents and is based on the user's criteria for relevance to the query. The second is the set of documents actually returned by the system as the result of the query. Now *precision* and *recall* can be defined as follows:

Precision is the ratio of the number of relevant images retrieved to the total number of images retrieved. Perfect precision (100%) means that all retrieved images are relevant.

$$precision = \frac{|relevant \bigcap retrieved|}{|retrieved|} \tag{4.1}$$

Recall is the ratio of the number of relevant images retrieved to the total number of relevant images. Perfect recall (100%) can be obtained by retrieving the entire collection, but the precision will be poor.

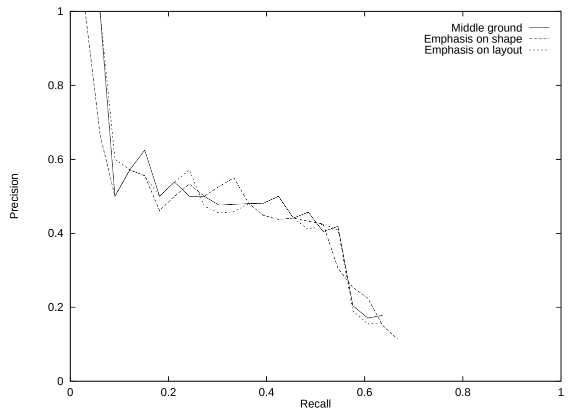
$$recall = \frac{|relevant \bigcap retrieved|}{|relevant|} \tag{4.2}$$

An IR system can be characterized in terms of performance by constructing a precision–recall graph for each query by incrementally increasing the size of the retrieved set, i.e., by measuring the precision at different recall points. Usually, the larger the retrieved set, the higher the recall and the lower the precision. This is easily done in MARS since the query processing algorithms are implemented as a pipeline.

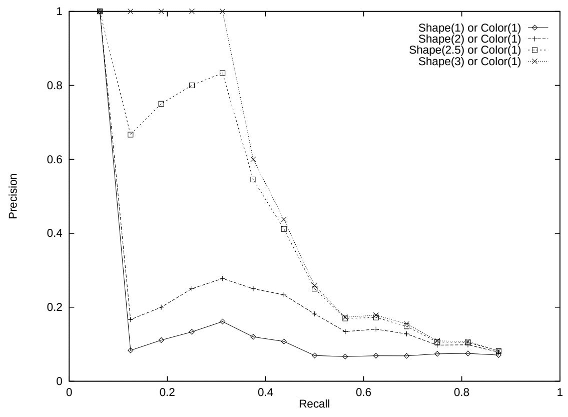
4.5.2 Data Sets

We have conducted experiments on two datasets. The first dataset is a collection of images of ancient artifacts from the Fowler Museum of Cultural History. We used a total of 286 images of such artifacts. The relevance judgments for this collection were obtained from a class project in Library and Information Science department at the University of Illinois. Experts in librarianship consulted with the curator of the collection to determine appropriate queries and their answers. Queries posed to this collection range from simple single feature queries to complicated queries involving all the operators described above and both retrieval models, namely fuzzy and probabilistic. In all, five groups of related images were chosen. For each group several queries involving single features and arbitrary operations between them as well as different weightings were constructed. These relevant query groups ranged in their cardinality from 9 to 33 images.

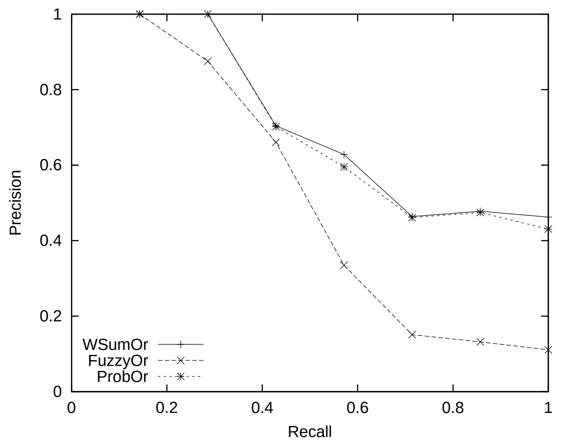
The second dataset is the Corel image collection dataset available from the UCI KDD repository. This collection contains around 70,000 images mostly of natural scenes. All images are cataloged into broad categories and each image carries an associated description. In this case, manually separating the collection into relevant and nonrelevant sets was infeasible due to the size. Instead we made use of our results in [113] to automatically determine the appropriate result set of each query. The dataset includes multidimensional image feature vectors for color histogram and texture data. We describe these features and their associated similarity predicates in appendix A.



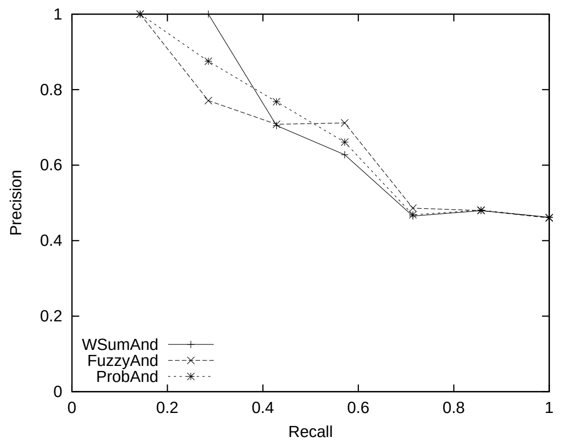
a) Complex query with different weights



b) Effects of varying the weighting on a query

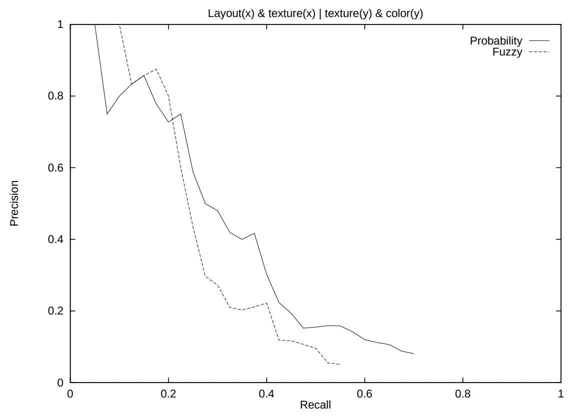


c) All Models for *or* query

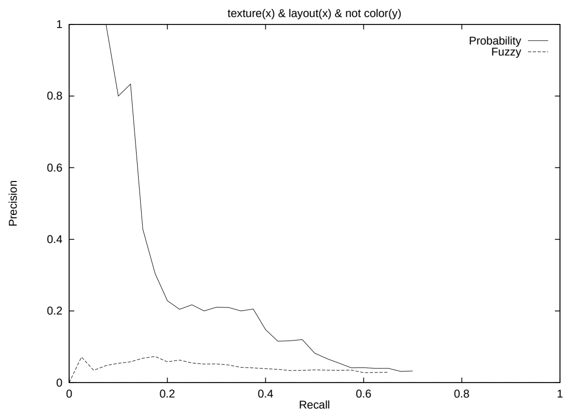


d) All Models for *and* query

Figure 4.2: Experimental Result Graphs



a) Comparing probability and fuzzy models



b) Comparing probability and fuzzy models

Figure 4.3: Corel Query Experimens

4.5.3 Results

4.5.3.1 Fowler Collection

In this section, we describe the results of some experiments performed on the image collection from the Fowler Museum. Since the complete set of experiments are too large to include, we present only the results of certain representative experiments.

Figure 4.2(a) shows a complex query (shape(I_i) *and* color(I_i) *or* shape(I_j) *and* layout(I_j) query) with different weightings. The three weightings fared quite similar, which suggests that complex weighings may not have a significant effect on retrieval performance.

We conducted experiments to verify the role of feature weighting in retrieval. Figure 4.2(b) shows results of a *shape or color* query, i.e., to retrieve all images having either the same shape or the same color as the query image. We obtained four different precision recall curves by varying the feature weights. The retrieval performance improves when the shape feature receives more emphasis.

We also conducted experiments to observe the impact of the retrieval model used to evaluate the queries. We observed that the weighted summation, fuzzy and probabilistic interpretations of the same query yields different results. Figure 4.2(c) shows the performance of the same query (a *texture or color* query) in the two models. We used the same query but with a conjunction to compare the performance of the retrieval models. The result is shown in Figure 4.2(d).

4.5.3.2 Corel Collection

For the Corel dataset, figure 4.3a) shows a query involving two database objects ranked under both fuzzy and probabilistic models. In this query, all terms are positive literals. Although at first the fuzzy model has good performance, the probabilistic model soon improves and stays better. Figure 4.3b) shows a query involving a negation. In this case the fuzzy model performed well below the probabilistic model. The next section discusses possible explanations.

4.5.4 Analysis of Data

Note the graphs shown are not always monotonic. The precision is expected to monotonically decrease as more and more images are retrieved. The small peaks in the graphs imply that a sequence of relevant images was quickly retrieved following a possibly long sequence of nonrelevant images. Taking averages over several queries would help in smoothing out these peaks. However, we do not take averages to depict the peculiar effects of individual queries. The way to read these graphs is that a higher curve is better than a lower one. This would mean that at all recall points, the precision was better.

We observe from Figure 4.2(b) that the weighting of features can improve performance dramatically. The weights for the queries were determined subjectively and several combinations were tried. We also observed (from Figure 4.2(a)) that complex weighting strategies may not always improve performance significantly.

We observed that the probabilistic model is superior to the fuzzy model for Corel queries. A possible explanation for this (specially for figure 4.3(b)) is that the *min* and *max* operations used in the fuzzy model are too restrictive. They take into account only one of all their parameters while the probabilistic operators take into account all the parameters. These results scaled from 286 to 70,000 images. This gives us confidence in the robustness of our approach.

Chapter 5

Similarity Query Processing

5.1 Introduction

This chapter discusses how we evaluate queries with complex Boolean similarity conditions. We support three mechanisms for generating the ranking of queries – the first is based on a weighted summation interpretation, the second is based on a fuzzy interpretation of similarity and the third is based on a probabilistic interpretation (see chapter 4).

In the discussion below, we will use the following notation. Attributes of tables are denoted by $a_1, a_2, \ldots, a_m$, and query values are denoted by $v_1, v_2, \ldots, v_t$. During query evaluation, each similarity predicate $p_i = a_j \sim= v_k$ is used to rank tuples in the table based on similarity. Thus, the predicate $p_i = a_j \sim= v_k$ can be thought of being a list of tuples ranked based on the similarity of v_k to all instances of a_j. A similarity expression e is either a single similarity predicate, or other similarity expressions combined with Boolean operators. We view a query condition $C(p_1, p_2, \ldots, p_n)$ over similarity predicates p_i as a tree whose leaves correspond to similarity predicates, and internal nodes correspond to similarity Boolean operators, therefore, subtrees represent similarity expressions. Specifically, non–leaf nodes are of one of three forms: $\wedge(e_1, e_2, \ldots, e_n)$, a positive conjunction of similarity expressions; $\wedge(e_1, e_2, \ldots, e_r, \neg e_{r+1} \ldots, \neg e_n)$, a conjunction consisting of both positive and negative similarity expressions; and $\vee(e_1, e_2, \ldots, e_n)$, which is a disjunction of positive similarity expressions. Notice that we do not consider an unguarded negation or a negation in the disjunction (that is, $r \geq 1$), since it does not make much sense. Typically, a very large number of entries will satisfy a negation query virtually producing the universe of the collection [14]. We therefore allow negation only when it appears within a conjunctive query to rank an entry on the positive feature discriminated by the negated feature. The following is an example of a similarity Boolean query condition: $C(p_1, p_2) = (a_1 \sim= v_1) \wedge (a_2 \sim= v_2)$ is a conjunction of two similarity predicates. Thus, the condition represents the desire to retrieve tuples whose attribute a_1 matches the value v_1 and its attribute a_2 matches the value v_2. Figure 5.1 shows an example query condition $C(p_1, p_2, p_3, p_4) = ((a_1 \sim= v_1) \wedge (a_2 = v_2)) \vee ((a_3 = v_3) \wedge \neg(a_4 = v_4))$ in its tree representation.

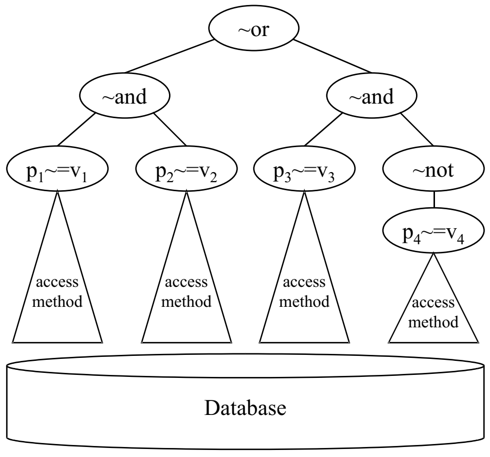


Figure 5.1: Sample Query Tree

5.1.1 Finding the Best N Matches

While our retrieval model provides a mechanism for computing a similarity of match for all tuples given a query, for the approach to be useful, techniques must be developed to retrieve the best N matches efficiently without having to rank each tuple. Such a technique consists of two steps:

- retrieve tuples in rank order based on each similarity predicate
- combine the results for single predicates to generate an overall result for the entire query

The first step is discussed in section 5.1.2. The second step is elaborated in section 5.1.3 for the background and in sections 5.2 for the weighted summation model, 5.3 for the fuzzy model, and 5.4 for the probabilistic model.

To merge the ranked lists, a query condition $C(p_1, p_2, \ldots, p_n)$ viewed as a tree is evaluated as a *pipeline* from the leaves to the root. Each node in the tree exposes to its parent a ranked list of tuples, with the ranking based on the similarity score. We say it exposes, because the key point is that each node produces the next best result strictly *on demand*. The algorithms to form the node's result depend upon the retrieval model used.

5.1.2 Predicate Evaluation

Predicates in the query condition usually correspond to a selection operation on a single attribute. For example, in figure 5.1, the leaf nodes correspond to selection operations based on the predicates $a_1 \sim= v_1, a_2 \sim= v_2, a_3 \sim= v_3$ and $a_4 \sim= v_4$, where a_i is an attribute in table, and v_i is its corresponding query value. This selection corresponds to ranking the table on the similarity between values in the column of attribute a and the query value v. A single–predicate selection operation *iteratively* returns the tuple whose corresponding value next best matches the given query value v. A simple way to implement the selection operation is a sequential file scan over the table followed by sorting. However, the I/O cost of the sequential scan operation increases linearly with the size of the table and hence may be expensive for large databases. The efficiency of the predicate evaluation can be improved by using appropriate indexing mechanisms that support nearest neighbor search. MARS has focussed on multidimensional vectors that can represent image features, therefore we focus on multidimensional indexing techniques. Several indexing mechanisms suited for multidimensional vectors or features have been proposed (e.g., Hybrid tree [25], R-trees [66], R+-trees [141], R*-trees [12], k-d-B-trees [129], hB-trees [44], TV-trees [95], SS-trees [167], vp-trees [164], M-trees [35].) Any such indexing mechanism can be used for indexing the vectors. These indices support efficient "nearest–neighbor" traversal to quickly generate the best matches in rank order given a query vector. In chapter 8 we revisit the nearest neighbor problem in the context of query refinement.

While we have discussed predicates as selections, they can also represent similarity join operations when a predicate p_i is of the form $p_i = a_j \sim= a_k$, that is, it involves two attributes usually from two different tables. This does not affect the problem at hand since the ranked list generated from a join predicate simply is over tuples that originate from two tables. The resulting ranked list has the same form as that of a similarity selection predicate and can readily be used with our algorithms. Similarity join algorithms that can generate such a list have been studied elsewhere [17, 87, 4, 117, 5, 146, 142, 69, 3, 104]. Of particular interest is the algorithm presented in [69] which is incremental. In chapter 8 we revisit this problem in the context of query refinement.

In this chapter, we concentrate on developing techniques for evaluating Boolean expressions over similarity predicates (for selection and join predicates), and assume the presence of a mechanism to provide efficient support for nearest neighbor search and join over multidimensional data and hence ranked retrieval at the predicate level.

5.1.3 Background on Evaluation Algorithms

This section defines some background concepts we use in the following sections. As described in chapter 3, a query produces a ranked list of tuples based on the similarity score of each tuple t to the query condition. Our evaluation model is as follows:

- Each node N in the condition tree returns a list of $\rho_i = \langle t.tid, score^i_{Q_N}(t)\rangle$ to its parent where:
  - $i = 1, 2, \ldots, n$ is the sequence number in which the ρ's are returned and n is the number of tuples
  - Q_N is the query condition subtree rooted at node N
  - $score^i_{Q_N}(t)$ is the similarity value of tuple t to the sub-query rooted at Q_N
  - for any two $\rho_i = \langle t.tid, score^i_{Q_N}(t)\rangle$, and $\rho_k = \langle t'.tid, score^k_{Q_N}(t')\rangle$ if $i < k$ then $score^i_{Q_N}(t) \geq score^k_{Q_N}(t')$ holds; that is, any ρ returned as an answer for the sub-query Q_N will have higher similarity than any pair returned later for the same sub-query, so ρ's are returned in sorted order by similarity

- Evaluation of an expression rooted at a node Q_N produces a sequence of ρ's. A cursor is maintained in this sequence to support the concept of current element; this sequence with cursor is called a *stream*.
- The notion of *best element* of a stream at any point is defined as the next $\rho = \langle t.tid, score_{Q_N}(t) \rangle$ that can be obtained from a stream satisfying the above criteria.
- A *stream* of ρ's supports the following operations:
  - *PeekNext* (or just *Peek*) that returns the *best element* of the stream without removing it from the stream
  - *GetNext* that returns the *best element* of the stream and removes it from the stream
  - *Probe(t.tid, Q_N)* that performs random access to tuple t and returns a $\rho = \langle t.tid, score_{Q_N}(t) \rangle$, that is, the tuple id and similarity pair corresponding to tuple t based on the expression rooted at node Q_N; not all operators require this support, we define it for all as a convenience
- The weights are not necessarily incorporated into the algorithms: the similarity scores are adjusted to reflect the weight when tuples are returned to the parent node. Since all weights are monotonic transformations, this does not alter the sequence of the results.

The algorithms presented in the following sections assume binary operators. n-ary operators can be implemented by either nesting binary operators (using the associativity property) or extending the algorithms to cope with n input streams. Extension of binary to n-ary operators is straightforward in all cases.

Given that the operators discussed are binary, and the inputs are streams as defined above, we can create a two dimensional representation where each axis corresponds to similarity values from one stream. Figure 5.2 depicts such a scenario. In this figure, the horizontal axis corresponds to stream A and the vertical axis to stream B. Points on this graph correspond to tuples whose similarity in stream A defines its A-axis coordinate and the similarity in B defines its B-axis coordinate. For instance, the point shown corresponds to tuple t with similarity values a' and b' in the respective streams.

Since streams are traversed in rank order of similarity, we obtain coordinates in sorted order from each stream. In the figure, a and b show the current similarity values of the best element currently in the streams (the cursor contents). Since all tuples from stream A with similarity values in the range [a,1] and all from stream B with similarity values in the range [b,1] have been read already, we can construct a rectangle bounded by the points (a,b) and (1,1) such that for all tuples in the rectangle, the similarity values corresponding to both streams have been observed. We refer to this rectangle as the *Observed Area Bounded Box* (OABB). Another interpretation of OABB is that it is the current intersection of the tuples observed so far in both streams. Projecting OABB onto the A axis yields another rectangle (called πA) that contains only tuples whose A coordinate

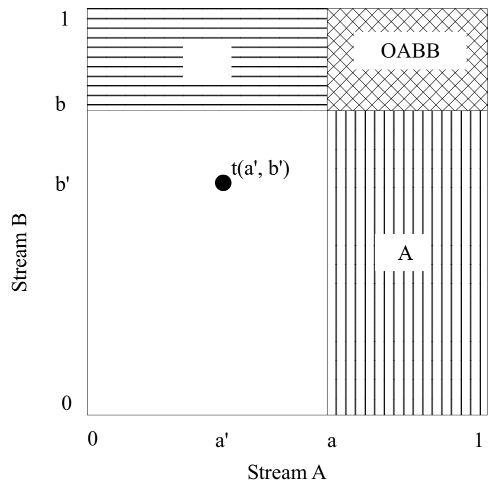


Figure 5.2: A Sample OABB Rectangle

is known, but its b coordinate is unknown; OABB and πA do not overlap. The same is true for the projection of OABB onto the B axis (the rectangle is called πB). $\pi A \cup \pi B$ denotes the tuples of which we have partial knowledge of their location in this 2-d space (i.e. exactly one co-ordinate known). Thus any tuple of which we have complete knowledge (both similarity values seen) must lie in OABB.

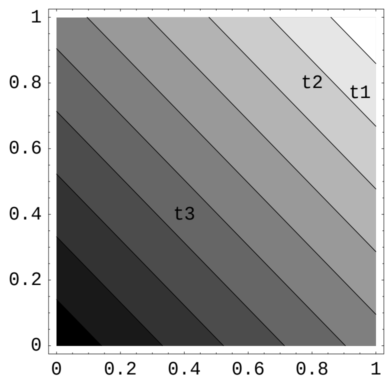
The following sections make use of these definitions to explain the functioning of the algorithms.

5.2 Weighted Summation Model Evaluation Algorithms

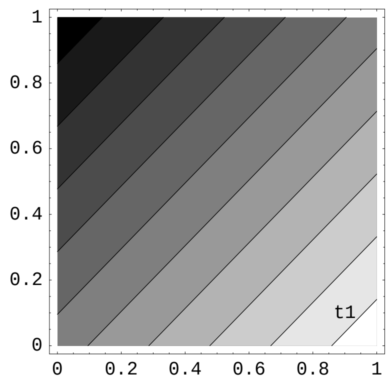
In this section we present the algorithms to evaluate similarity expressions for the weighted summation retrieval model presented in section 4.2. For simplicity we restrict ourselves to compute only binary nodes. That is, we assume that the query node Q has exactly two children, A, and B. We develop algorithms only for the following three cases: $Q = A \wedge B$, $Q = A \wedge \neg B$ and $Q = A \vee B$. As described in section 5.1, we only develop algorithms for positive conjunctive, negated conjunctive queries with a positive term and disjunctive queries.

In describing the algorithms we use the following notation:

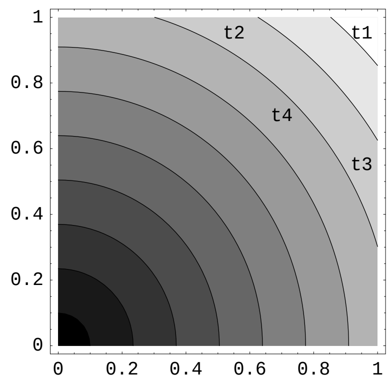
- A tuple t is represented by a pair of components $\langle t.tid, score_Q(t) \rangle$, composed by the key $(t.tid)$ which identifies the tuple id, and the score which identifies the tuple's similarity to the query expression $(t.score)$.
- A and B are assumed to be streams as defined in section 5.1.3.
- Associated with each query expression Q are three sets S_a, S_b and S_{res}. Initially each of these sets are empty. The query expression Q extracts tuples from the child streams (that is, A and B) and may buffer them into S_a and S_b (these represent the πA and πB rectangles from



a) Weighted summation *and* operator



b) Weighted summation *and not* operator



c) Weighted summation *or* operator

Figure 5.3: Contour Graphs for Weighted Summation Operators– Whiter is Higher, Darker is Lower Similarity Value

figure 5.2 respectively). The set S_{res} acts as a buffer of the tuples for the query expression Q. Once a query expression Q establishes the similarity score of tuple t for Q (that is, $score_Q(t)$), it places t in S_{res}(the result set.) Thus, $t.score$ refers to the similarity between tuple t and the query expression Q.

The following three sections describe the algorithms used to implement the above shown operations in an efficient manner. For clarity purposes, when describing the algorithms below we omit some critical error and boundary checking which needs to be considered in any implementation.

5.2.1 Conjunctive Query with Positive Subqueries

The algorithm in figure 5.4 computes the set of tuples ranked on their similarity score to the query $Q = A \wedge B$, given input streams A and B which are ranked based on their similarity score of tuples to A and B.

The operation performed in a binary operator node can be viewed as a function $S(x \in [0,1], y \in [0,1]) \rightarrow [0,1]$. As an aid to explain the algorithm, we use contour plots that show the value of $S(x,y)$. These plots depict lines along which the value of S is the same over different parameters, so called iso–similarity curves. In reality there are infinitely many such curves; the figures only show a few. The highest values of S (degree of membership) are in the white areas, the darker the region, the lower the value. Figure 5.3a) shows the plot that corresponds to the weighted summation *and* operator.

Imagine an overlay of Figure 5.2 on top of Figure 5.3a). As we traverse the streams A and B in similarity order, OABB grows and whole iso–similarity curves become completely contained in the OABB. Given the geometry of the curves, we notice that for any OABB defined as the rectangle bounded by (a,b)-$(1,1)$, there is a curve of minimum similarity along the line with value $c = w_a \times a + w_b \times b$, where w_a and w_b are the corresponding weights. Tuples contained in this OABB have been "seen", and their final similarity score is fully determined. Among these, tuples that lie on any iso–similarity curve completely enclosed in the OABB can be returned safely. Conversely, tuples in the OABB that lie on iso–similarity curves that intersect the OABB must be retained in an auxiliary set. Tuples in this auxiliary set become safe to return when the OABB covers a sufficiently low iso–probability curve such that its score is lower or equal to that of the now safe tuple. As an example, consider figure 5.3a). There are three tuples in the whole collection. t_2 is the first to be included in an OABB since streams A and B are explored in similarity order. When this happens, t_1 is partially known in πA. Even though OABB contains only one tuple with known final probability, it cannot yet be returned since it does not lie on an iso–probability curve completely covered by OABB. Then t_1 will be included in OABB, at this point, the OABB is large enough to cover t_1 and t_2, and we return t_1 as the best tuple. As the OABB grows to include t_3, it is large enough to ensure that t_2's iso–similarity curve does not intersect the OABB, so we return t_2 as the next best tuple. Finally, we return t_3.

We make use figure 5.5 to explain this algorithm. The figure shows one of many iso–similarity lines that correspond to the line $w_A \times A + w_b \times B = c$ where c is an arbitrary constant. The algorithm explores A and B in rank order, the next best score from A is a' and from B is b'. We want to find c such that the "safe" area is as large as possible and the line is completely enclosed in the rectangle formed by $(a', b') - (1, 1)$. For this to happen, the iso–similarity line must cross the OABB at either $(a', 1)$ or $(1, b')$, we evaluate the similarity score for both of these points and set c to the larger one. The algorithm follows this by evaluating the threshold c and if it has a tuple t in the OABB (S_{aux}) it checks to see if it is safe to return it. If ($t.score > c$) it returns t as the next results. Otherwise, the algorithm explores A and B taking from the higher valued stream first and trying to do a match with observations from the other stream. When a match occurs, it computes the final score and puts the tuple in S_{aux}. This process continues until there is a tuple that can be returned.

5.2.2 Conjunctive Query with Negative Subquery

We next develop the algorithm for computing the query $Q = A \wedge \neg B$; it is shown in figure 5.6. The algorithm is different compared to the one developed for the conjunctive query with no negative subquery. A strategy similar to the conjunctive query is possible if traversing the stream B in reverse order were possible. This implies a furthest neighbor query that is not supported. We use the positive expression to guide the search, and the negative expression to determine the final similarity score. We use *Probe* to determine the similarity score for the negative expression. As an example, tuple t_1 in figure 5.3 is best if it is located early in stream A and its similarity to the query expression that corresponds to B is very low.

This algorithm follows the safety criteria specified in the previous section, however only the stream for A is used in computing the similarity score of tuples according to $A \wedge \neg B$. Tuples are retrieved from the input stream A in rank order. For a given tuple t its similarity score with respect to the sub-query $\neg B$ is evaluated by performing a probe on tuple t and evaluating its similarity

```
function GetNextAnd_Wsum($A$, $B$)
// returns: next best tuple in $A$ and $B$
flag = TRUE
while (flag)
  $t_a$= Peek ($A$), $t_b$= Peek ($B$)
  c=max($w_A\times$ $t_a$.score+$w_B$, $w_A + w_B\times$ $t_b$.score)
  if $S_{aux} \neq \emptyset \wedge$ c < MaximumScore($S_{aux}$) then
    $t$= tuple from $S_{aux}$ with maximum score
    $S_{aux} = S_{aux} - t$
    flag = FALSE
  else
    if $t_a$.score > $t_b$.score then
      $t_a$= GetNext($A$)
      $S_a$= $S_a \cup$ $t_a$
      if $t_a \in S_b$ then
        $t_b$= tuple from $S_b$ equivalent to $t_a$
        $t$= $t_a$
        $t$.score = $w_A\times$ $t_a$.score + $w_B\times$ $t_b$.score
        $S_a$= $S_a$-$t_a$, $S_b$= $S_b$-$t_b$, $S_{aux} = S_{aux} \cup t$
      end if
    else
      // symmetric code to then branch
    end if
  end if
end while
$S_{res}$= $S_{res}\cup$ $t$
return $t$
end function
```

Figure 5.4: Algorithm that Implements the *and* Operator for the Weighted Summation Interpretation

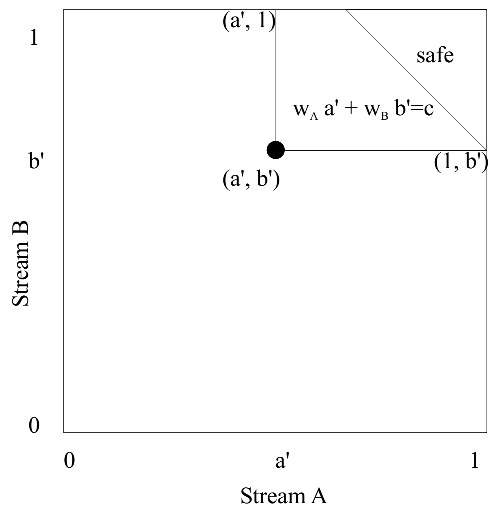


Figure 5.5: Threshold Computation for Weighted Summation Conjunction

```
function GetNextAnd_Not_Wsum($A$, $B$)
  // returns: next best tuple in $A$ and not $B$
  flag = TRUE
  while (flag)
    $t_a$= Peek ($A$) // best from $A$
    $c = w_A \times t_a$.score$+w_B$ // best possible, unknown B
    if $S_{aux} \neq \emptyset \wedge c <$ MaximumScore($S_{aux}$) then
      $t$= tuple from $S_{aux}$ with maximum score
      $S_{aux} = S_{aux} - t$
      flag = FALSE
    else
      $t$= GetNext($A$) // Consume from $A$
      $t_b$.score = Probe($t, \neg B$)
      $t$.score = $w_A \times t$.score $+ w_B \times t_b$.score
      $S_{aux} = S_{aux} \cup t$
    end if
  end while
  $S_{res}$= $S_{res} \cup t$
  return $t$
end function
```

Figure 5.6: Algorithm that Implements the *and not* Operator for the Weighted Summation Interpretation

score. Once the similarity score of a tuple t according to $\neg B$ is established, we determine its final score according to the query expression Q, and the tuple is inserted into an auxiliary set that is used to verify the safety criteria. A tuple is only returned if it successfully passes the safety test $t.score > c$, thus every returned tuple was in the auxiliary set. Effectively, every tuple retrieved from A results in a probe to B.

5.2.3 Disjunctive Query

Finally, to compute a disjunctive query node, we need the algorithm shown in figure 5.8. Disjunctive queries are hard to compute in this case. Consider figure 5.3c), tuples t_2, t_3 and t_4 have very similar similarity scores. Notice that in general there are some iso–similarity curves that will never be contained in any OABB (unless everything is read in). Since tuple t may have a higher similarity score in one stream than another, we would need to store it until a possibly much worse (and much later) match occurs from the other stream. Indeed, to return t_4, both t_2 and t_3 need to be first included in the OABB. Potentially, this results in a very large initial overhead (latency) to find the first few results.

Due to the particular exponent we chose for this interpretation of disjunction we have a unique shape for the space of this algorithm: a series of concentric circles[1]. We will use figure 5.7 to explain our algorithm. Given that the contour curves are circles we observe that for any circle with radius less than 1, we need to explore the entire inputs to give a definite answer for a tuple. For contour lines with a radius of more than 1, this is not so. Our algorithm concentrates on growing the "safe" area by exploring the streams A and B in rank order. We test the maximum possible similarity score c for the current OABB against the set S_{aux} of candidate tuples, if a tuple exists

[1]Note that we could have chosen higher exponents, in the limit, with an L_∞ metric, we have the fuzzy interpretation and can use that algorithm.

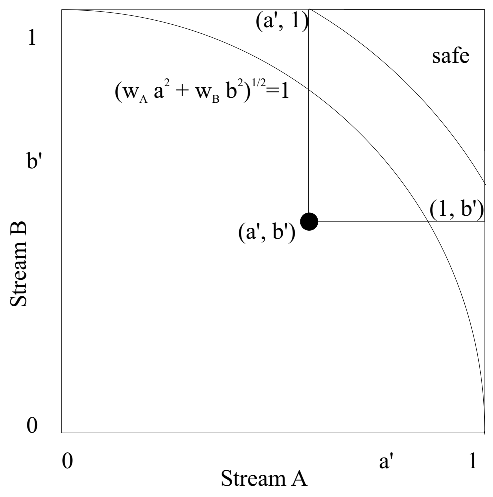


Figure 5.7: Threshold Computation for Weighted Summation Disjunction

with a higher score than c, we return it. Otherwise, we extract the next tuple from the stream with a higher top score and try to find a match in our cache of the other stream. If there is a match, we compute the final similarity score for the tuple and place it in S_{aux}.

The performance of this algorithm will degrade if we request many answers and the result score drops below 1 since, as observed above, this requires examining the entire inputs. We can modify this algorithm to offer optimistic guesses when the performance drops significantly. We describe how this can be done in our approach for the probabilistic disjunction which suffers from the same problems and the weighted summation disjunction.

5.3 Fuzzy Model Evaluation Algorithms

In this section, we present the algorithms to compute similarity expressions using the fuzzy interpretation presented in section 4.3. For simplicity we restrict ourselves to compute only binary nodes. That is, we assume that the query expression Q has exactly two children, A, and B. Algorithms are presented for the following three cases: $Q = A \wedge B$, $Q = A \wedge \neg B$ and $Q = A \vee B$. As described in section 5.1, we only develop algorithms for positive conjunctive, negated conjunctive queries with a positive term and disjunctive queries.

In describing the algorithms we use the following notation:

- A tuple t is represented by a pair of components $\langle t.tid, score(t)\rangle$, denoted by the tuple id ($t.tid$) and the degree of membership ($t.score$). The key identifies the tuple id and the degree of membership describes the similarity of match between the query feature and the database entries.

- A and B are assumed to be streams as defined in section 5.1.3.

```
function GetNextOr_Wsum($A$, $B$)
// returns: next best tuple in $A$ or $B$
flag = TRUE
while (flag)
  $t_a = \text{Peek}(A), t_b = \text{Peek}(B)$
  $c = \max(\sqrt{w_A \times a'^2 + w_B}, \sqrt{w_A + w_B \times b'^2})$
  if $S_{aux} \neq \emptyset \wedge c < \text{MaximumScore}(S_{aux})$ then
    $t$ = tuple from $S_{aux}$ with maximum score
    $S_{aux} = S_{aux} - t$
    flag = FALSE
  else
    if $t_a$.score > $t_b$.score then
      $t_a$= GetNext($A$) // remove best tuple from A
      $S_a$= $S_a \cup$ $t_a$
      if $t_a \in S_b$ then
        $t_b$= tuple from $S_b$ equivalent to $t_a$
        $t$= $t_a$
        $t$.score = $w_A \times$ $t_a$.score + $w_B \times$ $t_b$.score
        $S_a$= $S_a$-$t_a$, $S_b$= $S_b$-$t_b$, $S_{aux} = S_{aux} \cup t$
      end if
    else
      // symmetric code to then branch
    end if
  end if
end while
$S_{res}$= $S_{res} \cup$ $t$
return $t$
end function
```

Figure 5.8: Algorithm that Implements the *or* Operator for the Weighted Summation Interpretation

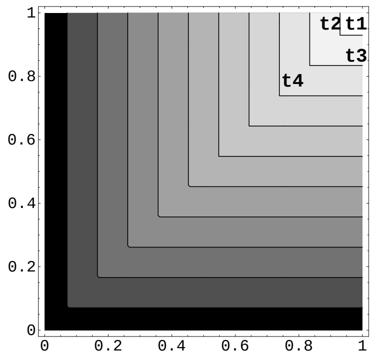
- Associated with each query expression Q are three sets S_a, S_b and S_{res}. Initially each of these sets are empty. The algorithm extracts tuples from the streams (that is, A and B) and may buffer them into S_a and S_b (these represent the πA and πB rectangles from figure 5.2 respectively). The set S_{res} acts as a buffer of the tuples for the query expression Q. Once a query expression Q is able to establish the degree of membership of tuple t for Q (that is, $degree_Q(t)$), it places t in S_{res}(the result set.) Thus, $t.score$ refers to the degree of membership of t according to Q, where $t \in S_{res}$.

The following three sections describe the algorithms. For clarity purposes, when describing the algorithms we omit some critical error and boundary checking which needs to be considered in an implementation.

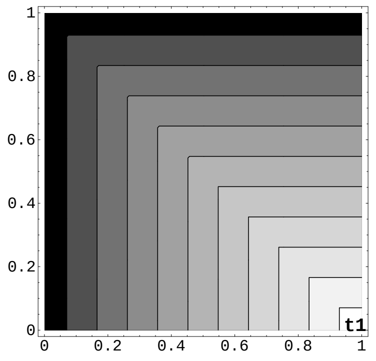
5.3.1 Conjunctive Query with Positive Subqueries

The algorithm shown in Figure 5.10 computes the list of tuples ranked on their degree of membership to the query expression $Q = A \wedge B$, given input streams A and B which are ranked based on the degree of membership of tuples in A and B.

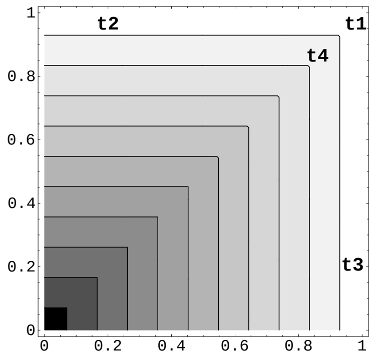
The operation performed in a binary operator node can be viewed as a function $S(x \in [0,1], y \in [0,1]) \rightarrow [0,1]$. As an aid to explain the algorithm, we use contour plots that show the value of $S(x,y)$. These plots depict lines along which the value of S is the same over different parameters, so called iso–similarity curves. In reality there are infinitely many such curves; the figures only



a) Fuzzy *and* operator



b) Fuzzy *and not* operator



c) Fuzzy *or* operator

Figure 5.9: Contour Graphs for Fuzzy Operators – Whiter is Higher, Darker is Lower Similarity Value

show a few. The highest values of S (degree of membership) are in the white areas, the darker the region, the lower the value. Figure 5.9a) shows the plot that corresponds to the fuzzy *and* operator.

Imagine an overlay of Figure 5.2 on top of Figure 5.9(a). As OABB grows, whole iso–similarity curves are completely contained in OABB. Given the geometry of the curves, we notice that for any OABB defined as the rectangle bounded by (a, b)-$(1, 1)$, there is a curve of minimum similarity along the square (c, c)-$(1, 1)$ where $c = \max(a, b)$. Tuples contained in this square are completely determined and are safe to be returned as answers. As an example, t_1 is contained in the first such square to appear. This is indeed the best tuple. Discriminating between t_2 and t_3 is more difficult. They both yield similar degrees of membership. Once the OABB has grown to contain both tuples, a decision as to the ranking is done. t_4 does not participate in this process since t_2 and t_3 are definitely better than t_4. The algorithm relies on this fact, but grows the OABB by exactly one tuple at a time, thus the next lower iso–similarity curve is exposed and the latest tuple to join OABB is the next answer. At each stage, the best tuple out of the sources A and B is chosen and added to sets S_a (πA) and S_b (πB) which function as buffers of tuples already observed from the corresponding stream. When a tuple is found that was already observed in the other stream, the loop terminates and this is the next best tuple according to the query expression Q (it just joined the rectangle OABB, thus encompassing the next iso–similarity curve that has an tuple). Notice that $\mid S_a \cup S_b \mid$ will never exceed the size of the feature collection. The resulting tuple is returned with the degree equal to the minimum degree of the tuple in both streams and lastly recorded in the result set.

```
function GetNextAnd_Fuzzy($A$, $B$)
1  returns: next best tuple in $A$ and $B$
2  while (TRUE)
3    $t_a$= Peek ($A$), $t_b$= Peek ($B$)
4    if $t_a.score > t_b.score$ then
5      $t_a$= GetNext($A$)
6      $S_a = S_a \cup t_a$
7      if $t_a.tid \in S_b$ then // tuple already seen in $B$
8        $t_b$= tuple $S_b[t_a.tid]$
9        exit loop
10     end if
11   else
12     if $t_b.score > t_a.score$ then
13       $t_b$= GetNext($B$)
14       $S_b = S_b \cup t_b$
15       if $t_b.tid \in S_a$ then // tuple already seen in $A$
16         $t_a$= tuple $S_a[t_b.tid]$
17         exit loop
18       end if
19     end if
20   end if
21 end while
22 // reached upon finding a common tuple in $S_a$ and $S_b$
23 $t.tid = t_a.tid$
24 $t.score = \min(t_a.score, t_b.score)$
25 $S_a = S_a - t_a, S_b = S_b - t_b, S_{res} = S_{res} \cup t$
26 return $t$
end function
```

Figure 5.10: Algorithm Returning the Next Best for the Fuzzy *and* Interpretation

5.3.2 Conjunctive Query with Negative Subquery

We next present the algorithm for computing the query $Q = A \wedge \neg B$; it is presented in Figure 5.11. Figure 5.9b) shows the contour plot that corresponds to this query. A strategy similar to the previous section could be used if traversing the stream B in reverse order was possible. This implies a furthest neighbor query that is not supported. The positive term is used to guide the search and the negative sub-query used to determine the final degree of membership. The OABB thus only considers entries from stream A and never grows in the B stream (which is never constructed). *Probe* is then used to complete the degree of membership of an tuple. As an example, tuple t_1 is best if it is located early in stream A and its similarity to the query feature that corresponds to B is very low.

This algorithm contains an auxiliary set S_{aux} to hold tuples retrieved from stream A and whose final degree of membership is established, but resulted lower than the membership degree in A. These tuples need to be delayed until such time that it is safe to return them. For each iteration of the loop, there are three possibilities:

- $S_{aux} \neq \emptyset \wedge \text{Peek}(A).score \leq \text{MaximumDegree}(S_{aux})$ the best tuple in the auxiliary set has higher membership degree than than the top tuple from A. In this case, the result is clear (return top tuple form S_{aux}), since *min* is used, no better tuple will come from A.
- $(S_{aux} = \emptyset \vee \text{Peek}(A).score > \text{MaximumDegree}(S_{aux})) \wedge \text{Peek}(A).score \leq \text{Probe}(\text{Peek}(A).tid, \neg B).score$ there is no better candidate on hold and the degree of the best tuple from A is lower (and

```
function GetNextAnd_Not_Fuzzy($A$, $B$)
1  // returns: next best tuple in $A$ and not $B$
2  while (TRUE)
3    $t_a$= Peek ($A$)
4    if $S_{aux} \neq \emptyset \wedge t_a.score <$ MaximumDegree($S_{aux}$) then
5      $t$ = tuple from $S_{aux}$ with maximum degree
6      $S_{aux} = S_{aux} - t$
7      exit loop
8    else
9      $t_a$= GetNext($A$) // consume from $A$
10     $t_b.tid = t_a.tid$
11     $t_b.score$ = Probe($t_a$, $\neg B$)
12     if $t_a.score \leq t_b.score$ then
13       $t = t_a$
14       exit loop
15     else
16       $S_{aux} = S_{aux} \cup t_b$
17     end if
18   end if
19 end while
20 $S_{res} = S_{res} \cup t$
21 return $t$
end function
```

Figure 5.11: Algorithm Returning the Next Best for the Fuzzy *and not* Interpretation

thus determines the answer) than the probe on the negative sub-query. The answer is the best tuple from A.

- $(S_{aux} = \emptyset \vee \text{Peek}(A).score > \text{MaximumDegree}(S_{aux})) \wedge \text{Peek}(A).score > \text{Probe}(\text{Peek}(A).tid, \neg B).score$ there is no better candidate on hold and the degree of the best tuple from A is higher than the probe on the negative sub-query. The final membership degree is determined by the probe and the tuple is sent to the auxiliary set to wait until it is safe to return it.

The loop iterates until a result is found.

5.3.3 Disjunctive Query

The algorithm shown in Figure 5.12 computes the set of tuples ranked on their degree of membership to the query $Q = A \vee B$, given input streams A and B which are ranked based on the degree of membership of tuples in A and B.

Figure 5.9c) shows the contour plot for the disjunctive fuzzy operator. By overlaying Figure 5.2 on Figure 5.9c) it can be seen that any OABB intersects iso–similarity curves (unless it is the whole space). This means no curve will be contained in any OABB, so unless the whole collection is retrieved, no definite ranking exists. This results in two options, 1) return only those tuples in the OABB, and 2) follow a different strategy. In the first case, to return t_2, the OABB would cover most of the collection, including t_4, but t_4 which is in OABB much earlier than any of t_2 or t_3 is worse than t_1, t_2 and t_3. Fortunately, we can follow a different strategy instead. By exploiting the properties of the *max* operator, t_1, t_2 and t_3 have the same membership degree, they only rely on one (the maximum) of their membership degrees in sub-queries and thus can safely ignore the other.

```
function GetNextOr_Fuzzy($A$, $B$)
1  // returns: next best tuple in $A$ or $B$
2  flag = TRUE
3  while (flag)
4    $t_a$= Peek ($A$), $t_b$= Peek ($B$)
5    if $t_a.score > t_b.score$ then
6      $t$= GetNext($A$) // remove best tuple from A
7    else
8      $t$= GetNext($B$) // remove best tuple from B
9    end if
10   flag = FALSE // assume candidate found
11   if $t.tid \in S_{res}$ then
12     flag = TRUE // find new, was returned
13   end if
14 end while
15 $S_{res} = S_{res} \cup t$
16 return $t$
end function
```

Figure 5.12: Algorithm Returning the Next Best for the Fuzzy *or* Interpretation

Since tuples with better membership degrees are examined first, this is sufficient to determine the final membership degree.

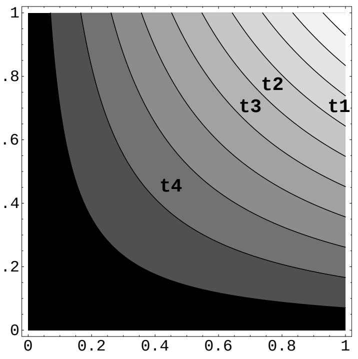
The algorithm essentially consists of a merge based on the degree of membership value but makes sure that a tuple that was already returned is ignored as a result (duplicate removal). This accomplishes the desired *max* behavior of the degree function associated with the disjunction in the fuzzy model.

5.4 Probabilistic Model Evaluation Algorithms

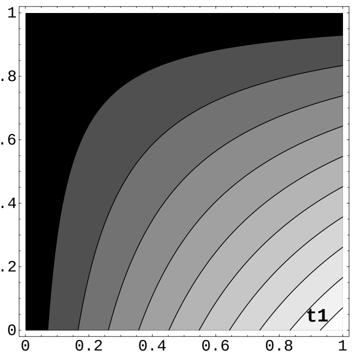
In this section, we present the algorithms to evaluate expressions for the probabilistic retrieval model. For simplicity we restrict ourselves to compute only binary nodes. That is, we assume that the query node Q has exactly two children, A, and B. As for the fuzzy model, algorithms are only developed for the following three cases: $Q = A \wedge B$, $Q = A \wedge \neg B$ and $Q = A \vee B$.

In describing the algorithms the following notation is used:

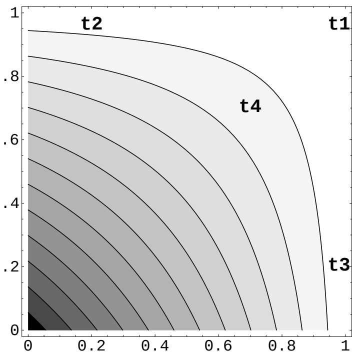
- A tuple t is represented by a pair of components $\langle t.tid, score_Q(t)\rangle$, composed by the key ($t.tid$) which identifies the tuple id, and the similarity which identifies the probability that the tuple satisfies the query ($t.score$).
- A and B are assumed to be streams as defined in section 5.1.3.
- Associated with each query expression Q are three sets S_a, S_b and S_{res}. Initially each of these sets are empty. The query expression Q extracts tuples from the child streams (that is, A and B) and may buffer them into S_a and S_b (these represent the πA and πB rectangles from figure 5.2 respectively). The set S_{res} acts as a buffer of the tuples for the query expression Q. Once a query expression Q is able to establish the probability of match of tuple t for Q (that



a) Probabilistic *and* operator



b) Probabilistic *and not* operator



c) Probabilistic *or* operator

Figure 5.13: Contour Graphs for Probabilistic Operators – Whiter is Higher, Darker is Lower Similarity Value

is, $probability_Q(t)$), it places t in S_{res}(the result set.) Thus, $t.score$ refers to the probability that tuple t matches the query expression Q.

The following three sections describe the algorithms used to implement the above shown operations in an efficient manner. For clarity purposes, when describing the algorithms below we omit some critical error and boundary checking which needs to be considered in any implementation.

5.4.1 Conjunctive Query with Positive Subqueries

The algorithm in figure 5.14 computes the set of tuples ranked on their probability of match to the query $Q = A \wedge B$, given input streams A and B which are ranked based on their matching probability of tuples in A and B.

It is interesting to note that an algorithm similar to the one proposed in section 5.3.1 will not work properly. To understand this, observe figure 5.13a) and recall the OABB suggested in section 5.1.3. The rectangle will contain a region with tuples that have been observed in both streams, yet the distribution of probability is complex within this rectangle. This requires a modified algorithm that returns tuples only when it is safe to do so. Similarly to the fuzzy case, there is a minimum value iso–similarity curve completely covered by any OABB. The probability value for this curve is defined by its intersection with the axes. So, for an OABB bounded by (a, b)-$(1, 1)$, all tuples with known probability of more than the maximum of a and b are safe to be returned. Note however that the OABB will also contain tuples with known final probability less than this amount, these are retained in an auxiliary set. Tuples in this auxiliary set become safe to return when the OABB covers a sufficiently low iso–probability curve such that its probability is lower or equal to that of the now safe tuple. As an example, consider figure 5.13a). There are four tuples in the whole collection. t_2 is the first to be included in an OABB. When this happens, t_1 is partially known in πA. Even though OABB contains only one tuple with known final probability, it cannot yet be returned since it does not lie on an iso–probability curve completely covered by OABB. Then t_1 will be included in OABB, but it also cannot yet be returned. The curve just below t_1 intersects with a vertical line drawn from t_3. Until this is cleared, t_1 and thus t_2 cannot be returned. When t_4 is added, the highest iso–probability curve that is lower than t_1, t_2 and t_3 is clear of the projection of t_4 onto the axes, thus, it is safe to return all of t_1, t_2 and t_3 at this stage.

The algorithm first tests if there is a safe tuple in the auxiliary set to return and does so if there is one. Otherwise, it extracts the next best tuple from the better of A or B and tries to include it in OABB by finding it to be in the intersection. If unsuccessful, it is stored in one of the sets corresponding to πA or πB. The loop iteratively checks for safety and fetches tuples until a safe tuple can be returned. Note that unlike in the fuzzy case, the only way to exit the loop is by an tuple being safe as defined above. Of course in the fuzzy algorithms, returned tuples were also safe, but the safety criteria is so simple, that multiple loop exists exist.

An optimization on this algorithm is to slightly modify the safety criteria. The criteria described above is simple to understand: a tuple is not safe until all the region of higher probability has been seen. The danger of not following this strategy is that for some tuples, only one probability has been retrieved, and the other is unknown. The above safety criteria is pessimistic in that it assumes that the other probability could be any value, while it is in fact bounded by the top probability in the stream where the tuple has not yet been retrieved. If t_k.score requires $t_k.\text{score}_A$ and $t_k.\text{score}_B$ to compute $t_k.\text{score} = t_k.\text{score}_A \times t_k.\text{score}_B$, then an upper bound on the probability of tuple t_k is:

$$\begin{aligned} &\text{Peek}(A).\text{score} \times t_k.\text{score}_B && \text{if } t_k.\text{score}_B \text{ is known, or} \\ &\text{Peek}(B).\text{score} \times t_k.\text{score}_A && \text{if } t_k.\text{score}_A \text{ is known} \end{aligned} \tag{5.1}$$

This more sophisticated criteria is not incorporated in figure 5.14, instead the simpler criteria described above is included.

5.4.2 Conjunctive Query with Negative Subquery

We next develop the algorithm for computing the query $Q = A \wedge \neg B$; it is shown in figure 5.15. The algorithm is different compared to the one developed for the conjunctive query with no negative sub-query. As described for the fuzzy model, a similar method to the conjunctive query with only positive sub-queries could be used if traversing the B stream in inverse was feasible. This is however not the case. This algorithm follows the safety criteria specified in the previous section, however only the stream for A is used in computing the probability of tuples according to $A \wedge \neg B$. Tuples are retrieved from the input stream A in rank order. For a given tuple t its probability with respect to the sub-query $\neg B$ is evaluated by performing a probe on tuple t and evaluating its probability of match. Once the probability of match of a tuple t according to $\neg B$ has been established, we can determine its final probability according to the query Q, and the tuple is inserted into an auxiliary

```
function GetNextAnd_Probability($A$, $B$)
  // returns: next best tuple in $A$ and $B$
  flag = TRUE
  while (flag)
    $t_a$= Peek ($A$), $t_b$= Peek ($B$)
    if $S_{aux} \neq \emptyset \wedge$ Max($t_a$.score, $t_b$.score) <
            MaximumProbability($S_{aux}$) then
      $t$= tuple from $S_{aux}$ with maximum probability
      $S_{aux} = S_{aux} - t$
      flag = FALSE
    else
      if $t_a$.score > $t_b$.score then
        $t_a$= GetNext($A$)
        $S_a$= $S_a \cup t_a$
        if $t_a \in S_b$ then
          $t_b$= tuple from $S_b$ equivalent to $t_a$
          $t$= $t_a$
          $t$.score = $t_a$.score $\times$ $t_b$.score
          $S_a$= $S_a$-$t_a$, $S_b$= $S_b$-$t_b$, $S_{aux} = S_{aux} \cup t$
        end if
      else
        // symmetric code to then branch
      end if
    end if
  end while
  $S_{res}$= $S_{res} \cup t$
  return $t$
end function
```

Figure 5.14: Algorithm that Implements the *and* Operator for the Probabilistic Interpretation

set that is used to verify the safety criteria. A tuple is only returned if it successfully passes the safety test, thus every returned tuple was in the auxiliary set. Effectively, every tuple retrieved from A results in a probe to B.

5.4.3 Disjunctive Query

Finally, to compute a disjunctive query node, we need the algorithm shown in figure 5.16. Disjunctive queries are hard to compute in this case. Consider figure 5.13c), tuples t_1, t_2 and t_3 have very similar probabilities. In the fuzzy case, the iso–similarity curves were parallel to the axes and we could exploit the max behavior. This is not possible here. In addition notice that no iso–probability curve will be contained in any OABB (unless everything is read in). Two distinctions exist with the fuzzy version,

- the final probability does depend on all the query terms while in the fuzzy model, only the best one is relevant
- iso–probability curves are not even piecewise parallel to the axes

Since tuple t may have a higher probability in one stream than another, we would need to store it until a possibly much worse (and much later) match occurs from the other stream. Indeed, to return t_1, both t_2 and t_3 need to be included in the OABB. Potentially, this results in a very large initial overhead (latency) to find the first few results. To overcome this limitation, once a tuple is seen for the first time, its full probability is established with appropriate probes.

```
function GetNextAnd_Not_Probability($A$, $B$)
  // returns: next best tuple in $A$ and not $B$
  flag = TRUE
  while (flag)
    $t_a$= Peek ($A$) // best from $A$
    if $S_{aux} \neq \emptyset \wedge t_a$.score < MaximumProbability($S_{aux}$) then
      $t$= tuple from $S_{aux}$ with maximum probability
      $S_{aux} = S_{aux} - t$
      flag = FALSE
    else
      $t$= GetNext($A$)
      $t_b$.score = Probe($t, \neg B$)
      $t$.score = $t$.score $\times$ $t_b$.score
      $S_{aux} = S_{aux} \cup t$
    end if
  end while
  $S_{res}$= $S_{res} \cup$ $t$
  return $t$
end function
```

Figure 5.15: Algorithm that Implements the *and not* Operator for the Probabilistic Interpretation

```
function GetNextOr_Probability($A$, $B$)
  // returns: next best tuple in $A$ or $B$
  flag = TRUE
  while (flag)
    $t_a$ = Peek($A$), $t_b$ = Peek($B$)
    if $S_{aux} \neq \emptyset \wedge t_a$.score + $t_b$.score − $t_a$.score $\times$ $t_b$.score $\leq$
            MaximumProbability($S_{aux}$) then
      $t$ = tuple from $S_{aux}$ with maximum probability
      $S_{aux} = S_{aux} - t$
      flag = FALSE
    else
      if $t_a$.score > $t_b$.score then
        $t_a$= GetNext($A$)
        $S_a$= $S_a \cup$ $t_a$
        if $t_a \notin S_b$ then // do a probe
          $t_b$ = Probe($B, t_a$.id)
          $t.tid = t_a.tid$
          $t$.score = $t_a$.score + $t_b$.score − $t_a$.score $\times$ $t_b$.score
          $S_{aux} = S_{aux} \cup t$
        end if
      else
        // symmetric code to then branch
      end if
    end if
  end while
  $S_{res}$= $S_{res} \cup$ $t$
  return $t$
end function
```

Figure 5.16: Algorithm that Implements the *or* Operator for the Probabilistic Interpretation

To follow the algorithm, the notion of safety is used again. When is it safe to return t_1 given that we only have partial knowledge for t_2 and t_3? Probes are used to establish missing probabilities and a final probability score is computed. Tuples are then stored into an auxiliary set until they can safely be returned.

Tuples can safely be returned when their known probability is larger than the best to come. All tuples in S_{aux} can be partitioned into those with probability above (*safe set*) and below (*unsafe set*) the value $Peek(A).\text{score} + Peek(B).\text{score} - Peek(A).\text{score} \times Peek(B).\text{score}$. Those in the *safe* partition necessarily have higher probability than those in the *unsafe* partition, but also any combination of tuples that remain to be considered in streams A and B would fall into the current *unsafe* partition. Tuples from the safe set can now be returned in rank order. The algorithm grows S_{aux} one by one and at each stage verifies for safety. The safe set may contain at most one element, if present it is returned as an answer and removed from the safe set.

The algorithm assumes that probing is possible on sub-queries. So far, only algorithms based on negation have required this and then only for the negation operator. If probing on sub-queries is expensive, an alternate algorithm (not shown here) can be constructed as in the conjunctive query case. When one component probability of an tuple t_k is known, an upper bound on the final probability is established by:

$$\begin{aligned}
upper(t_k) = &\quad \text{Peek}(A).\text{score} + t_k.\text{score}_B - \text{Peek}(A).\text{score} \times t_k.\text{score}_B \quad \text{if } t_k.\text{score}_B \text{ is known,} \\
upper(t_k) = &\quad \text{Peek}(B).\text{score} + t_k.\text{score}_A - \text{Peek}(B).\text{score} \times t_k.\text{score}_A \quad \text{if } t_k.\text{score}_A \text{ is known}
\end{aligned} \tag{5.2}$$

And the known probability component is a lower bound ($lower(t_k)$). Based on the known bounds for t_k, instead of waiting to complete its final probability, it is estimated as its lower bound ($lower(t_k)$). Once no upper bound ($upper(t_j)$) of any unsolved tuple can exceed $lower(t_k)$, and no combination of any tuples left in A and B can exceed $lower(t_k)$, then t_k is safe to return and is returned with probability $lower(t_k)$.

5.5 Comparison of Algorithms to Other Work

Recently, [46] proposed an algorithm to return the top k answers for queries with monotonic scoring functions that has been adopted by the Garlic multimedia information system under development at the IBM Almaden Research Center [42]. A function F is monotonic if $F(x_1, \ldots, x_m) \leq F(x'_1, \ldots, x'_m)$ for $x_i \leq x'_i$ for every i. Note that the scoring functions for both conjunctive and disjunctive queries for both the fuzzy and probabilistic Boolean models satisfy the monotonicity property. In [46], each stream i is accessed in sorted order based on the degree of membership to form a ranked set X_i, and a set $L = \cap_i X_i$ that contains the intersection of the objects retrieved from all streams. Once L contains k objects (to answer a top k query), all objects in $\cup_i X_i$ are used to perform probes on whichever streams they were not read from. This essentially completes all the information for the objects in the union and enables a final definite scoring and ranking of all objects in $\cup_i X_i$, then the top k are the final answer. This algorithms works in the general case, and is tailored in [46] to some specific specific scoring functions. This algorithm relies on reading a number of objects from each stream until it has k in the intersection. Then it falls back on probing to enable a definite decision. In contrast, our algorithms are tailored to specific functions that combine object scoring (here called fuzzy and probabilistic models). Our algorithms follow a *demand driven data flow* approach [60]. Instead of asking for the top k objects, only the next best element is requested and returned. This follows a fine grained pipelined approach. According to the cost model proposed in [46], the total database access cost due to probing can be much higher compared to the total cost due to sorted access. Only our algorithms involving negation require probing. We used probing in section 5.4.3 for convenience, but sketched an alternate algorithm that does not require probing. Our demand driven approach reduces the wait time of intermediate answers in a temporary file or buffer between operators in the query tree. This model is efficient in its time-space product memory costs [60]. On the other hand, in Garlic, the data items returned by each stream must wait in a temporary file until the completion of the probing and sorting process. In [126], the authors discuss how to best estimate the probability of disjunctions, their results are similar to ours. Also, in the query processing model followed in MARS, the operators are implemented as iterators which can be efficiently combined with parallel query processing [59]. In our work, we assumed that the execution costs of different predicates are roughly the same and can be ignored. In [29] the authors deal with the case where there this does not hold, and construct an approach to minimize probing of expensive predicates. In [108] the authors explore algorithms to merge ranked lists for complex user defined join predicates.

Another approach to optimizing query processing over multimedia repositories has been proposed in [32]. It presents a strategy to optimize queries when user's specify thresholds on the grade of match of acceptable objects as filter conditions. It uses the results in [46] to convert top-k queries to threshold queries and then process them as filter conditions. It shows that under certain conditions (uniquely graded repository), this approach is expected to access no more objects than the strategy in [46]. Like the former approach, this approach also requires temporary storage of intermediate answers and sorting before returning the answers to the user. Furthermore, while the above approaches have mainly concentrated on the fuzzy Boolean model, we consider the weighted summation, fuzzy, and probabilistic models.

5.6 Evaluation

Our evaluation addresses the performance of some algorithms discussed in this chapter. Evaluation of query processing and optimization focuses on the traditional computational performance as opposed to the quality of results.

For our experiments we used the Corel collection available online at `http://kdd.ics.uci.edu/`. This dataset provides color and texture features alongside other information for 68,040 of the Corel images. We populated a relation *CorelTable* with the attributes: *tuple_id* (integer to uniquely identify the image), *category* (single word stating the image category, e.g., "Wildlife"), *text_description* (description of image, up to 300 bytes long), *color* (16-dimensional color histogram), and *texture* (16-dimensional co-occurrence texture). We concentrated on retrieving those images closest to given examples in color and texture. The query condition q is thus $q = \widetilde{color} \ \sim\wedge \ \widetilde{texture}$ where $\widetilde{color} = (color \sim= c_1)$ and color histogram similarity is computed by histogram intersection, and $\widetilde{texture} = (texture \sim= t_1)$ where texture similarity is computed by Euclidean distance and converted to similarity (see appendix A). We use a *weighted summation* interpretation of similarity operators as presented in section 4.2, thus the score for each tuple t is computed as $t.score = w_{col} \times \widetilde{color}(t) + w_{tex} \times \widetilde{texture}(t)$, and $w_{col} + w_{tex} = 1$. The following indices exist on $CorelTable$: a B-Tree index on *category*, a 16-d multidimensional HybridTree [25] index on *color* and a 16-d multidimensional HybridTree index on *texture*. We generate the input ranked lists into our algorithm with a k-nearest neighbor algorithm (descibed in detail in chapter 8) on the HybridTree index. The relation and index page size is 4KB, and all experiments were conducted on a Sun Ultra Enterprise 450 with 1 GB of memory and several GB of secondary storage, running Solaris 2.7.

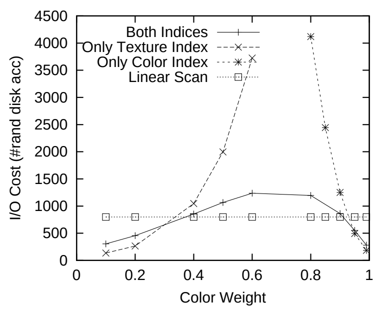
A naive evaluation is to do a sequential scan and evaluate the query condition for each tuple. We consider this as the baseline and compare the performance of several query plans for the query condition q presented above and then add a precise predicate to q. First, we consider a similarity query for the best 10 matches involving the query condition q for the four plans discussed in example 3.3.10. We compare their I/O and CPU performance as follows:

- *Linear Scan (LS)*, or $top_{10}(\sigma_{\widetilde{color} \ \sim\wedge \ \widetilde{texture}}(CorelTable))$ where the I/O cost is $\frac{68{,}040}{avg\_tuples\_per\_page} \approx$ 8000 sequential disk accesses since $avg\_tuples\_per\_page = \frac{page\_size}{tuple\_size} = \frac{4096}{450} = 9$. Assuming sequential I/O to be about 10 times faster than random I/O, in terms of random I/O the cost is $\approx$ 800. The CPU cost is the cost of computing the similarity score of each tuple in CorelTable.

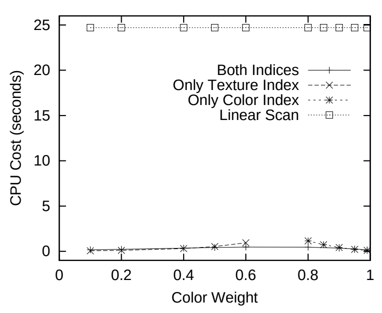
- *Color Index (COLIDX)*, or $top_{10}(\sigma_{\widetilde{color}_{IDX}}(CorelTable) \hat{\bowtie}_{\widetilde{color} \ \sim\wedge \ \widetilde{texture}} CorelTable)$ uses the color index to retrieve the best color matches and access full tuples in CorelTable for other attributes. The random disk I/O cost is that of accessing the index pages from the k-NN algorithm, and that of accessing full tuples in CorelTable; each full tuple access is one random disk access. The CPU cost is the time needed by the k-NN algorithm for index computations, and for scoring full tuples.

- *Texture Index (TEXIDX)*, or $top_{10}(CorelTable \ \hat{\bowtie}_{\widetilde{color} \ \sim\wedge \ \widetilde{texture}}(\sigma_{\widetilde{texture}_{IDX}}(CorelTable)))$. As above but using the texture index instead of the color index.

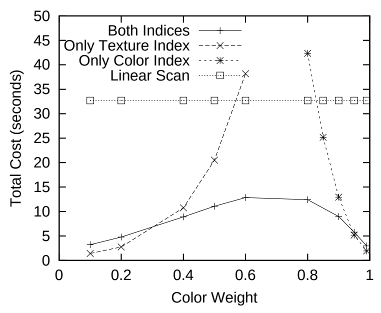
- *Both Indices (BOTHIDX)*, or $top_{10}(\sigma_{\widetilde{color}_{IDX}}(CorelTable) \hat{\bowtie}_{\widetilde{color} \ \sim\wedge \ \widetilde{texture}}(\sigma_{\widetilde{texture}_{IDX}}(CorelTable)))$, retrieves the best color and texture matches and merges them with our algorithm for weightes summation conjunction. I/O and CPU costs are obtained as above.



a) Random I/O Cost



b) CPU Cost



c) Total Cost

Figure 5.17: Performance of Different Relative Weighs of Color and Texture

The total cost (in seconds) is the addition of the CPU and the I/O costs (assuming 10ms per random I/O and 1ms per sequential I/O). The results are averaged over 50 queries with randomly generated query points for $\widetilde{color}$ and $\widetilde{texture}$.[2] Figure 5.17 compares the performance of the plans for various values of color weights w_{col} ($w_{tex} = 1 - w_{col}$). *TEXIDX* is best for low values of $w_{col} \leq 0.3$ while *COLIDX* is best for very high values of $w_{col} \geq 0.95$. The asymmetry between *TEXIDX* and *COLIDX* is due to differing similarity distributions in the feature spaces [13]. *BOTHIDX* is most efficient for all other values of $0.3 \leq w_{col} \leq 0.95$. *LS* suffers mainly due to its high CPU cost (figure 5.17(b)).

[2]Note that for *COLIDX*, *TEXIDX* and *BOTHIDX*, we implemented the priority queue-based k-NN algorithm on top of the multidimensional index structure [70, 131] as well as the "merging" algorithm that retrieves items from the index(es), accesses the full tuples, maintains them in sorted order based on the overall similarity score and incrementally returns the best retrieved tuple when it is guaranteed to be the next best match (i.e., no unexplored tuple with a higher score exists) [46, 115].

Chapter 6

Query Refinement Model

6.1 Introduction

The query refinement model is an important aspect of our approach which lets users *iteratively* pose and *refine* queries, and is intimately related to the data model and the similarity matching model. Figure 6.1 shows how query refinement is done in our model. The user forms an abstract *information need* that describes her desired information, expresses it in SQL form, submits the SQL query to the system which creates a plan and then executes it and sends the results back to the user. The user can then indicate to the system which results are indeed relevant and which results are not. Query Refinement is achieved by modifying a query condition in three ways: (1) changing the interpretation of the user-defined domain-specific similarity predicate functions (intra-domain), (2) changing the interpretation of operators that connect other predicates together (inter-domain), and (3) adding new predicates to the condition or removing subexpressions from it. The refined query is then re-optimized and executed, possibly re-utilizing previous results, and a new answer set is produced.

We now discuss the meaning of query refinement and present a framework for several levels of refinement. The discussion is independent of any particular interpretation of intra- and inter-domain predicates and operators. A possible interpretation is presented in chapter 7.

6.2 Semantics of Query Refinement

We first discuss the meaning of *query refinement* that we expect to realize in our model. Intuitively, we expect that changes in the query based on the users feedback will lead to better answers. Formally, the notion of *convergence* refers to the desire that at each iteration the results improve and eventually reach a stable state. For a specific query let there be:

- An arbitrarily ranked set, called the *Ground Truth Set*, provided by the user where the $t.score$ attribute reflects the degree to which the user judges the tuple to match the query and reflects the users subjective perception (*information need*). This ranked set is based on the entire possible set of tuples, i.e., the universe.

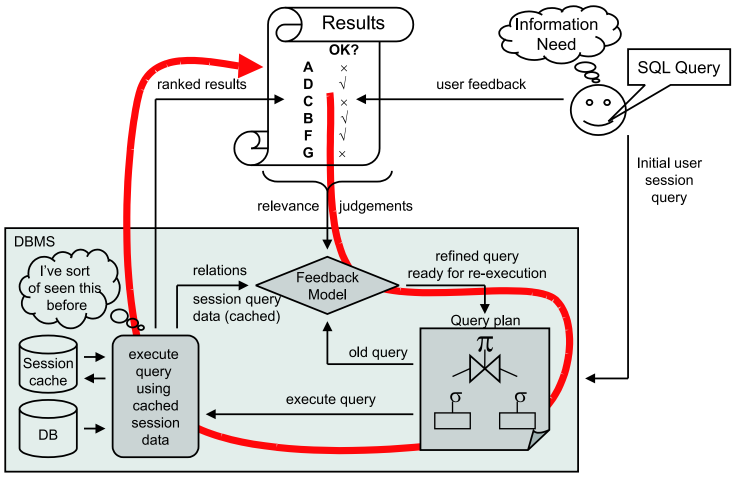


Figure 6.1: Query Refinement Architecture

- A response to the query; the user receives a ranked answer set of n tuples where the $t_i.score$ attribute is the similarity of tuple t_i to the query. This is the *Ranked Answer Set*.
- At each iteration, a user supplied set of tuples $t_i \in$ *Ranked Answer Set* and for each tuple a judgment for the degree to which it satisfies the query. This is the *Feedback Set*.
- A function $F(RS_1, RS_2)$ that compares two ranked sets and returns a similarity value for the comparison in the range [0,1] where 1 means the ranked sets are identical and 0 means they are very different. This can be the normalized recall metric [137] or a variation [124].

As discussed previously and shown in figure 6.1, the user executes an initial query and then submits a *Feedback Set* which is a series of tuples each with attached relevance judgments for relevance feedback, and a new ranked set is produced: $Ranked\ Answer\ Set_i \oplus Feedback\ Set \longrightarrow Ranked\ Answer\ Set_{i+1}$.[1]

The goal of query refinement is to change the original query such that the similarity between the *Ground Truth Set* and the *Ranked Answer Set* increases until they are as close as possible or identical. This property is known as *convergence*:

Definition 6.2.1 (Convergence) *Convergence of query refinement is the non-decreasing similarity between the* ground truth set *and the results with each iteration i:*
$F(Ranked\ Answer\ Set_{i+1}, Ground\ Truth\ Set) \geq F(Ranked\ Answer\ Set_i, Ground\ Truth\ Set)$.
■

The meaning of query refinement is that in response to user feedback, the ranked set answers produced by the system adapt to a ranked set that the user constructed based on the whole

[1]Assuming the $\oplus$ operator denotes the refinement operation.

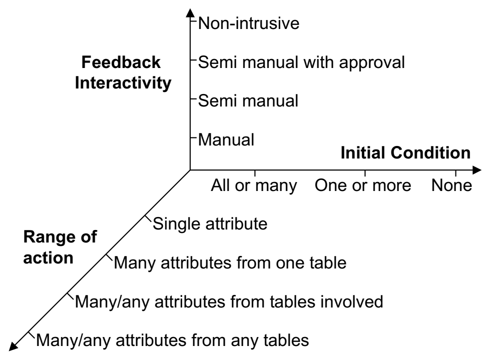

Feedback Interactivity
Non-intrusive
Semi manual with approval
Semi manual
Manual
Initial Condition
All or many
One or more
None
Range of action
Single attribute
Many attributes from one table
Many/any attributes from tables involved
Many/any attributes from any tables

Figure 6.2: Refinement Model Design Space

database. The assumptions of this model are that users are not malicious, that is, they provide feedback that is consistent with their desired goal or information need, and that it is possible to provide a Ground Truth Set for the entire universe.

6.3 Query Refinement Framework

Query Refinement spawns a large *design space*, including the modality in which a user specifies the feedback, the initial conditions of the query, and the scope of the refinement which can include a single or more attributes or tables. Figure 6.2 shows three dimensions of the identified design space. In this figure, the closer to the origin, the simpler it is for a system to compute the next query to execute. Further from the origin the system has more difficulty in determining the new query. This query refinement design space is general in that it only identifies different design aspects and does not propose a specific method.

Feedback Interaction. Along the top dimension, the user provides decreasingly explicit information about relevant items, by going from detailed per column feedback judgments to feedback on the overall tuple. The options are:

1. **Fully manual:** the user changes predicates, and possibly adds or removes tables. This is the traditional interaction with databases and search engines where users formulate new queries themselves.
2. **Column level:** the user indicates which attribute values, among the result tuples, are relevant to the user and which are not. The system then modifies the query to incorporate these preferences.
3. **Tuple level:** the user selects entire tuples for feedback and lets the system figure out how to modify the query.

Tuple level feedback is less specific than column level feedback as the system does not know which attribute values are relevant, only that the entire tuple is relevant. From a user interface standpoint, tuple level feedback is less burdensome to the user.

For example, a user searches for jobs that pay around \$60,000 in the *job openings* table of example 1.0.2, the answers are: $\langle programmer, New\ York, \$70,000\rangle$, $\langle tester, Los\ Angeles, \$65,000\rangle$, and $\langle analyst, Los\ Angeles, \$60,000\rangle$. The user focuses on jobs in California, therefore she indicates for the first tuple that New York is a bad location example, but \$70,000 is a good salary example (column level feedback). The second tuple is good overall as it pays more than expected and the job is in California (tuple level feedback). Finally, the last tuple's job is in California, a good example, and the salary is also good (column level feedback). The feedback summary is: $\langle -, bad, good\rangle$, $\langle tuple\ is\ good\rangle$, and $\langle -, good, good\rangle$.

Initial Conditions. The user can give many initial conditions, or give less and less predicates until only the tables involved are specified without a condition and the system is left to find the proper predicates. In increasing order of difficulty, the options are:

1. **Many or all:** The user specifies many or all the initial conditions desired, which implies the user has a very certain idea of what she wants.
2. **One or a few:** the user has some idea of what she wants, and allows initial ranking of the results. The user then tightens or widens the query using refinement, and the system adds or removes predicates to reflect this.
3. **No condition:** The user specifies only the tables involved in the query and the system presents a cross product of the tables. The user provides feedback on these answers and machine learning techniques can be used to learn appropriate predicates.

Following our above job search example, the original query included only a target salary of \$60,000, but no predicate on location. The query falls in the second category (*one or a few*) since later location became important. If the user knew in advance the importance of the job location, she would have included predicates on location and salary to fully specify all initial conditions. This illustrates the need to support addition and removal of predicates in query conditions.

Refinement Scope. In the bottom dimension, the *scope* of the refinement says what structural components are involved in refinement. The scope increases from refining a single predicate on an attribute to balancing multiple or complex attributes in multiple tables and adding or removing predicates in the query. This is depicted in figure 6.3, the different options include:

1. **Single predicate:** the interpretation of the similarity predicate for individual attributes change thus affecting the ranking of results.
2. **Multiple predicate balancing:** the contributions of different predicates are re-balanced in the scoring rule to emphasize important predicates.
3. **Query predicate addition/deletion:** predicates are added to or dropped from the query condition and the predicates in the scoring rule are re-balanced; this incorporates in the query those predicates that are important but not considered initially, or dismisses useless predicates from the query.

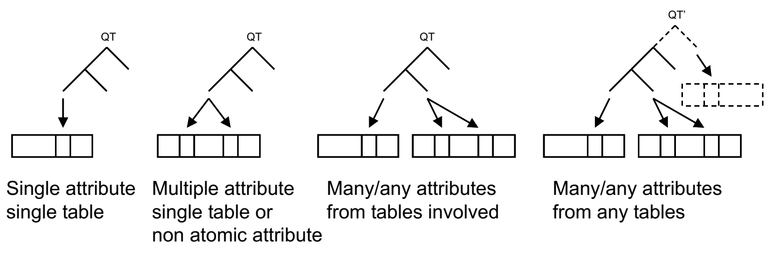

QT
QT
QT
QT'
Single attribute single table
Multiple attribute single table or non atomic attribute
Many/any attributes from tables involved
Many/any attributes from any tables

Figure 6.3: Feedback Range of Action

Several parts of the query condition may change in our above job search example. The original target salary of \$60,000 was improved by two examples, thus, the interpretation of the predicate that matches salary may change to a target salary of \$65,000 to reflect this (single-predicate refinement). We decided to add a predicate on location to concentrate on California, this is an example of *query predicate addition/deletion* and the relative importance among these two predicates must be balanced properly.

6.4 What can change?

As discussed in section 6.3, the design space for a refinement system is extensive, here we briefly list in the broadest terms what can change in a query. Using feedback, a query refinement can modify the following parameters to improve a query:

- change the structure of the query condition
- add or remove similarity predicates to the query condition
- change the weight associated with each similarity predicate
- change the parameters of any similarity predicate
- change the query values of any similarity preciate

Example 6.4.1 (Refining Job Searches) *In example 3.3.8, we matched applicants to positions based on skills and salary using the query: applicant* $\bowtie_{resume \sim= description\ \sim\wedge\ desired\_salary \sim= salary}$ *jobs. By examining the results, the user implicitly wants to look for senior applicants which have more skills, and considers that geographic proximity may be a benefit commensurate with salary. This query may be modified in the following ways:*

- *Add home_location* $\sim=$ *location to the query with a disjunction to show that geographic proximity is important but indicates that salary or geographic proximity are desirable and interchangeable.*
- *Add age* $\sim> 30$ *to indicate a desire for senior level applicants.*

- *Modify the interpretation of the $\sim\wedge$ operator to give more importance to the resume $\sim=$ description predicate. This is done by increasing the weight of the predicate.*

This produces the query:
$\sigma_{age\sim>30}(applicant) \bowtie_{resume\sim=description\ \sim\wedge\ (desired\_salary\sim=salary\ \sim\vee\ home\_location\sim=location} jobs.$
This list of changes falls into several categories, the user initially indicated a subset of the desirable conditions thus producing an initial ranking, and the changes involve attributes in several tables. ■

In this chapter we described our similarity retrieval model, specified what we expect from query refinement and gave a framework for classifying different features of a query refinement model. This chapter has ignored concrete approaches to query refinement. In chapter 7 we explore how query refinement interacts with similarity conditions.

Chapter 7

Query Refinement Strategies

7.1 Introduction

The purpose of query refinement is to better capture what the user wants, i.e., the users *information need*. The user forms an abstract *information need* that describes what she wants, expresses it in SQL form with similarity predicates and scoring rules, submits the SQL query to the system which evaluates it and sends the results to the user. Users then indicate good and bad answers and let the system modify the query in order to improve the quality (ordering) of answers.

Query refinement occurs at two levels, within predicates over the same domain through the user-defined type specific refinement functions, called *intra-predicate* query refinement, and across different attributes in a query, thus it affects the interpretation of the operators described in the similarity matching model described in chapter 3, this is called *inter-predicate* query refinement.

Many different refinement implementation are possible, in this section, we present a set of refinement techniques based on certain assumptions about the query conditions, in particular, we assume a weighted summation model to combine multiple similarity predicates and restrict conditions to conjunctions. The motivation for this restriction is that users rarely pose similarity queries looking for distinct concepts, which makes disjunctions less attractive. The specific interpretations provided here fall in the $\langle$ *manual* $-$ *semi manual*, *all or many conditions*, *single attribute* $-$ *many attributes* $\rangle$ category, in which (1) semi manual means the user submits relevance feedback via examples or a new query (manual) and (2) specifies most of the needed conditions, and single attribute to many attributes means that the techniques can simultaneously change the interpretation of individual intra-domain predicates and across domains, but is unable to add predicates to ranking expressions, thus it is not "many/ any attributes from tables involved". Even with this minimal expansion in the design space, much can be accomplished in refinement algorithms and retrieval performance improvement as demonstrated in the experiments (see section 7.5).

Figure 7.1 shows the many levels at which refinement is possible in this particular interpretation/scenario. Those parameters or knobs that can be tuned or modified are shown in boldface in the figure. We discuss this figure in a bottom-up fashion. In intra-predicate refinement, the similarity predicate for individual query points can be modified by changing the similarity functions

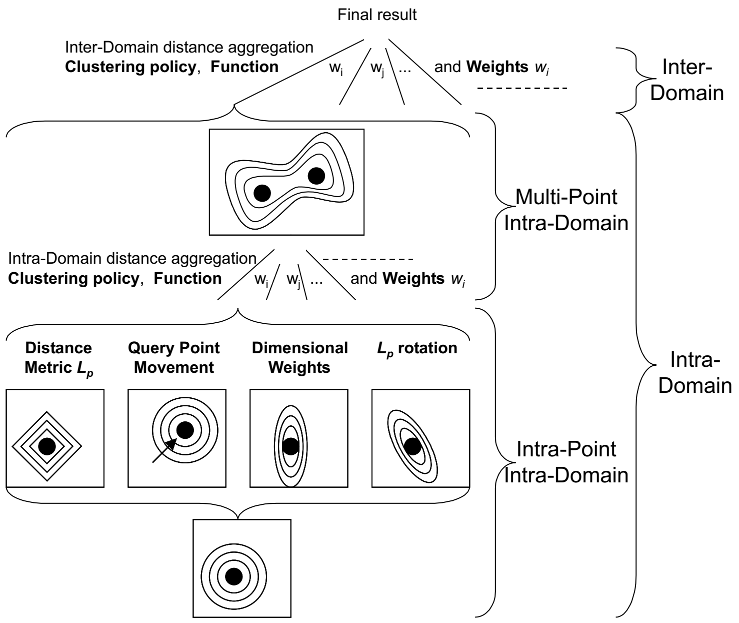

Final result
Inter-Domain distance aggregation
Clustering policy, Function
w_i w_j ... and Weights w_i
Inter-Domain
Multi-Point Intra-Domain
Intra-Domain distance aggregation
Clustering policy, Function
w_i w_j ... and Weights w_i
Distance Metric L_p
Query Point Movement
Dimensional Weights
L_p rotation
Intra-Domain
Intra-Point Intra-Domain

Figure 7.1: Feedback Parameters to Adjust for a Sample Two-dimensional Data-type — adjustable Parameters are in Boldface

itself,next the query point itself can be moved, the shape of the query region can be distorted with weights along type dimensions, and the shape of this warped query space can also be rotated as done by the MindReader system [134, 78]. Still in the intra-predicate refinement, multiple points are aggregated with a function θ which here is a weighted summation with individual weights for each query points similarity. The clustering policy determines how new points can be added or old points removed from the multi-point query. Finally several domains are combined in the inter-predicate refinement, here, a function θ is used to combine the inter-predicate similarity values, here we again use a weighted summation function [116]. The clustering policy can add or delete predicates from the ranking expression, in this case, it does neither and just follows a re-weighting of the predicates already in the ranking expression (cluster). Modifying the clustering policy to include new predicates would expand the scope of this scenario to include the "many/any attributes from tables involved" property from our framework.

Support for query refinement requires the DBMS to maintain sufficient state information and provide supporting interfaces to query refinement functions. In section 7.2 we present the needed supporting data structures and describe the interface and goals of the refinement support functions. These data structures are used by the refinement algorithms. Query refinement is achieved by modifying queries in three ways: (1) changing the interpretation and weights of query conditions (inter-predicate refinement), (2) adding or removing predicates to the condition (inter-predicate refinement), and (3) changing the interpretation of the user-defined domain-specific similarity predicate functions (intra-predicate refinement). The system provides algorithms to support inter-predicate refinement since it is independent of any domain, this does not mean that a single interpretation is used, rather multiple interpretations of refinement and scoring rules are possible within the system framework. On the other hand, user-defined functions are responsible for providing intra-predicate refinement. We discuss strategies for *inter-predicate* refinement in section 7.3 and discuss several strategies for a sample multidimensional user-defined data-type in section 7.4.

7.2 Query Refinement Processing Support

To support refinement of a result, we collect relevance judgments on a subset of the answers and generate an auxiliary table used by the refinement algorithms.

7.2.1 *Answer* and *Feedback* Tables

To refine a query, all values of tuples where the user gave feedback judgments must be available, they are needed since refinement is guided by the values of relevant and non-relevant tuples. In some queries, maintaining this information may be complicated due to loss of source attributes in projections or joins where attributes may be eliminated. To enable query refinement, we must include sufficient information in the answer to recompute similarity scores. Therefore, we construct an *Answer* table as follows:

Algorithm 7.2.1 (Construction of temporary *Answer* table for a query) *The* answer *table has the following attributes: (1) a tuple id (*tid*), (2) a similarity score S that is the result of evaluating the scoring rule, (3) all attributes requested in the select clause, and (4) a set H of hidden attributes. The set of hidden attributes is constructed as follows: for each similarity predicate in the query, add to H all fully qualified*[1] *attribute(s) that appear and are not already in H or the attribute was requested in the query select clause. All attributes retain their original data-type.* ■

Results for the hidden attributes are not returned to the calling user or application. Retaining attributes that are involved in scoring simplifies the detailed re-computation of scores later needed for the refinement process.

We also construct a *Feedback* table for the query as follows:

Algorithm 7.2.2 (Construction of temporary *Feedback* table for a query) *The attributes for the* Feedback *table are: (1) the tuple id (tid), (2) a* tuple *attribute for overall tuple relevance, and (3) all attributes in the select clause of the query. All attributes are of type integer (except the tid). The table is populated with the user's feedback.* ■

The user populates the feedback table with a variation of the SQL *insert* command:

insert into feedback tid=$\langle tid\rangle$ **relevance=**$\langle value\rangle$ **values** ($value_1$, . . . , $value_n$)

The parameters are (1) the *tuple id* of the tuple in the result ranked set for which a judgment is

[1]Same attributes from different tables are listed individually.

T:

tid	a	b	c	d

select S, a, b
from T
where $d > 0$ **and**
$b \sim= \hat{b}(v_b)[w_b]$ $\sim$ **and**
$c \sim= \hat{c}(v_c)[w_c]$
order by S **desc**

w_a, w_b, w_c	predicate weights
$\hat{a}, \hat{b}, \hat{c}$	query values for
v_a, v_b, v_c	predicate parameters

Answer:

tid	S	a	b	c
1	s_1	a_1	b_1	c_1
2	s_2	a_2	b_2	c_2
3	s_3	a_3	b_3	c_3
4	s_4	a_4	b_4	c_4

Feedback:

tid	tuple	a	b
1	1	0	0
2	0	0	1
3	0	-1	1
4	0	0	-1

Scores:

tid	$a \sim= a_1$	$b \sim= \hat{b}$	$c \sim= \hat{c}$
1	1.0	0.8	0.9
2	–	0.9	–
3	0.2	0.8	–
4	–	0.3	–

Figure 7.2: Data Structure Example for Query Refinement – Single Table

submitted, (2) an integer that specifies the tuple level relevance for tuple level feedback (the value immediately following the *relevance* keyword), and (3) an integer value for each column in the result ranked set specifying the column-level relevance. No feedback is provided for computed attributes such as the score. The integer values for the tuples and columns are 0 for neutral opinion, positive values (i.e., 1) for good matches, and negative values (i.e., -1) for bad matches.

Example 7.2.1 (Answer and Feedback table construction) *Figure 7.2 shows a table T with four attributes, and a query that requests only the score S, and the attributes a and b. The query has similarity predicates $b \sim= \hat{b}$, and $c \sim= \hat{c}$, so attributes b and c should be in H. But b is in the select clause, so only c is in H and becomes the only hidden attribute. Attribute d is not in H since the predicate $d > 0$ is not a similarity predicate. The feedback table is constructed as described in algorithm 7.2.2 and contains the tid, tuple, a, and b attributes. Figure 7.2 shows a sample Feedback table, notice that we used both tuple and attribute level feedback in this example. The user indicated that tuple 1 is a good answer overall, attribute b of tuples 2 and 3 are good examples, but attribute a of tuple 3 and b of tuple 4 are bad examples. There is no detailed feedback on attribute c since it is hidden from the user, but it is important since tuple 1 is relevant and thus the value of attribute c of tuple 1 is needed.*

Figure 7.3 shows Answer *and* Feedback *tables when a join involves a similarity predicate. The figure shows a join of tables R and S using the similarity predicate $R.b \sim= S.b$. Attribute a is not in H since it is listed in the select clause. Although the attributes R.b and S.b are not in the project list, we retain their values since they come from two different tables (the figure lists them together). We again keep these attributes hidden from the user. The Feedback table is constructed as above and populated: tuples 1 and 3 are overall relevant, while tuple 2 is not relevant overall. Attribute a of tuple 4 is also deemed not relevant. Note that it is not necessary for the user to give feedback for all tuples in the answer set despite what is shown in this example.* ■

R:

tid	a	b	c

S:

tid	b	c	d

select S, a, d
  from R, S
  where $d > 0$ **and**
    $a \sim= \hat{a}(v_a)[w_a]$ $\sim$ **and**
    $R.b \sim= S.b(v_b)[w_b]$
  order by S **desc**

w_a, w_b	predicate weights
$\hat{a}, \hat{b}$	query values
v_a, v_b	predicate parameters

Answer:

tid	S	a	d	$\langle$b$\rangle$
1	s_1	a_1	d_1	$R.b_1, S.b_1$
2	s_2	a_2	d_2	$R.b_2, S.b_2$
3	s_3	a_3	d_3	$R.b_3, S.b_3$
4	s_4	a_4	d_4	$R.b_4, S.b_4$

Feedback:

tid	tuple	a	d
1	1	0	0
2	-1	0	0
3	1	0	0
4	0	-1	0

Scores:

tid	$a \sim= \hat{a}$	$d \sim= d_1$	$R.b \sim= S.b$
1	0.7	0.9	0.8
2	0.8	0.5	0.7
3	0.3	0.4	0.6
4	0.6	–	–

Figure 7.3: Data Structure Example for Query Refinement – Similarity Join

7.2.2 *Scores* Table

We build an auxiliary *Scores* table that combines the user supplied feedback and the values from the answer table, and then populate the table:

Algorithm 7.2.3 (Construction of *Scores* table) *The Scores table has the following attributes: (1) a tuple id (tid), (2) all attributes in the select clause, (3) all attributes in the hidden set H. If two attributes appear in a similarity join predicate, they are fused into a single attribute. All attributes except the tid are real valued in the range [0,1] or are undefined. The table is populated with the scores of values for which the user specifies feedback and whose attributes are involved in some similarity predicate in the query. For a pair of values such as in a join predicate, a single score results. Figure 7.4 shows the algorithm to populate the* Scores *table.* ■

Example 7.2.2 (Populating the *Scores* table) *Consider for example, figure 7.2. The* Scores *table contains columns labeled $b \sim= \hat{b}$ and $c \sim= \hat{c}$, that correspond to attributes b and c in the query. An entry in the column for a tuple in the* Scores *table indicates the similarity score between the query values $\hat{b}$ and $\hat{c}$ and the values of attributes b and c in the corresponding tuple from the* Answer *table. Note that the* Scores *tables in figures 7.2 and 7.3 also contain columns labeled $a \sim= a_1$ and $d \sim= d_1$, the computation of those columns and their relevance in query refinement will be discussed in section 7.3.1.* ■

7.2.3 Support Functions

The refinement process is implemented by user-defined functions attached to our system. Refinement functions modify the similarity predicates interpretation. We define the interface for refinement functions for similarity predicates and scoring rules, section 7.4 discusses their implementation.

```
for-all tuples $t$ in Feedback table
  for-all attributes $x$ in $t$
    if $((t.x \neq 0) \vee (t.tuple \neq 0) \wedge \exists$ predicate $Z$ on $x$
      // there is non-neutral feedback on attribute $x$
      Scores table $[tid = t.tid, attribute = x]$ =
        Z(Answer table $[tid = t.tid, attribute = x]$)    // recreate detailed scores
```

Figure 7.4: *Scores* Table Computation

```
$applies(x)$ = predicates compatible with data-type($x$)
$x_{query}$ = Answer table $[top\ relevant\ tuple, attribute = x]$
for-all predicates $Z(x, x_{query}) \in applies(x)$
  for-all tuples $t$ in Answer table
    if Feedback table $[tid = t.tid, attribute = x] \neq 0$
      Scores table $[tid = t.tid, attribute = x]$ =
        Z(Answer $[tid = t.tid, attribute = x], x_{query}$)
  compute relevant and non-relevant avg and stddev
  if $avg\_rel - stddev\_rel > avg\_non\_rel + stddev\_non\_rel$
    keep predicate $Z$ and its difference
add predicate $Z$ with maximum difference to the query
```

Figure 7.5: Predicate Addition Computation

Definition 7.2.1 (Similarity predicate refinement function interface) *For each similarity function that implements a similarity predicate, there is a corresponding function named by appending "*_refine*" to its name. The function inputs are a list of answer values, a list of corresponding answer values in case this predicate is used in a join, and a list with the corresponding feedback judgments: $similarity\_predicate\_refine(\ list\langle value_{answer}\rangle,\ list\langle value_{join}\rangle,\ list\langle relevance\rangle)$. The function modifies the corresponding predicate query value(s), parameters, etc. to better reflect what the user wants. The $value_{answer}$ and relevance lists are taken directly from the* Answer *and* Feedback *tables. The $value_{join}$ list is taken from the* Answer *table if this is a join predicate, or is set to null otherwise.* ■

Example 7.2.3 (Refinement function for location similarity) *The function* close_to *implements geographic similarity for the data-type of example 3.3.2. The corresponding refinement function is declared as:* close_to_refine$(list\langle value_{answer}\rangle, list\langle value_{join}\rangle, list\langle relevance\rangle)$. ■

7.3 Inter-Predicate Query Refinement

First we discuss how to add and remove similarity predicates from conjunctions and then turn our attention to the problem of finding the appropriate weights for each similarity predicate. As discussed in section 7.1, we focus on conjunctions and the weighted summation model in developing these techniques.

7.3.1 Predicate Addition

The inter-predicate selection policy of figure 7.1 adds predicates to or removes them from the query condition, we discuss addition of predicates here and later their removal. Users may submit coarse initial queries with only a few predicates, therefore supporting the addition of predicates to a query is important. In figure 7.2, attribute a was ranked relevant for tuple 1 (since the tuple is relevant overall), and non-relevant for tuple 3. This suggests there might be merit in adding a predicate on attribute a to the query. We now discuss if and how this can be achieved.

Intuitively, we must be conservative when adding a new predicate to the query as it may significantly increase the cost of computing the answers, and may not reflect the users intentions. For these reasons, when adding a predicate, it must be added with a small initial weight to keep the addition from significantly disrupting the ranking of tuples since before the addition the effective predicate on the attribute was *true*.

To add a predicate on an attribute to the query, a suitable predicate must be found that applies to the given data-type, fits good the given feedback, and has sufficient support. Our algorithm for predicate addition is shown in figure 7.5. Based on catalog information, a list of similarity predicates that apply to the data-type of attribute a is constructed, this list is denoted by $applies(a)$. The *Answer* table is explored to find the highest ranked tuple that contains positive feedback on attribute a and take a's value. In figure 7.2, this value is a_1. This value (a_1) becomes the plausible query value for the new predicate. Next, we iterate through all predicates in the list $applies(a)$, and for each value of attribute a in the *Answer* table that has non-neutral feedback, we evaluate its similarity score using a_1 as the query value, and set the similarity score in the *Scores* table. The weights for the predicate are the default weights for that predicate (taken from the metadata). Say predicate $a \sim= a_1 \in applies(a)$ is under test. Under the *Scores* table in figure 7.2, $a \sim= a_1 = 1.0$ and $a \sim= a_3 = 0.2$ where a_1 is relevant and a_3 is not relevant. A predicate has a good fit if the average of the relevant scores is higher than the average of the non-relevant scores. In this case, this is true. In addition to this good fit, to justify the added cost of a predicate, we require that there is sufficient support. Sufficient support is present if there is significant difference between the average of relevant and non-relevant scores; this difference should be at least as large as the sum of one standard deviation among relevant scores and one standard deviation among non-relevant scores. If there are not enough scores to meaningfully compute such standard deviation, we empirically choose a default value of one standard deviation of 0.2. In this example thus, the default standard deviations for relevant and non-relevant scores add up to $0.2 + 0.2 = 0.4$ and since $average(relevant) - average(non\text{-}relevant) = 1.0 - 0.2 = 0.8 > 0.4$ then we decide that predicate $a \sim= a_1$ is a good fit and has sufficient support (i.e., separation) and is therefore a candidate to being added to the query. We choose $a \sim= a_1$ over other predicates in $applies(a)$ if the separation among the averages is the largest among all predicates in $applies(a)$.

The new predicate $a \sim= a_1$ is added to the query condition with a weight equal to one half of its fair share, i.e., $1/(2 \times |predicates\ in\ scoring\ rule|)$. We do this to compensate for the fact that we are introducing a predicate not initially specified and do not want to greatly disrupt the ranking. If there were four predicates before adding $a \sim= a_1$, then $a \sim= a_1$ is the fifth predicate and its fair share would be 0.2, we set a weight of $0.2/2 = 0.1$, and re-normalize all the weights in the condition as described in chapter 4.

7.3.2 Predicate Deletion.

A predicate is eliminated from the query condition if its weight falls below a threshold during re-weighting since its contribution becomes negligible, the remaining weights are re-normalized. For example, if we use *average weight* re-weighting (discussed in section 7.3.3) in figure 7.3, then the new weight for attribute a is: $\max(0, \frac{0.7+0.3-0.8-0.6)}{4}) = \max(0, -0.1) = 0$. Therefore, predicate $a \sim= \hat{a}$ is removed.

7.3.3 Predicate Re-weighting

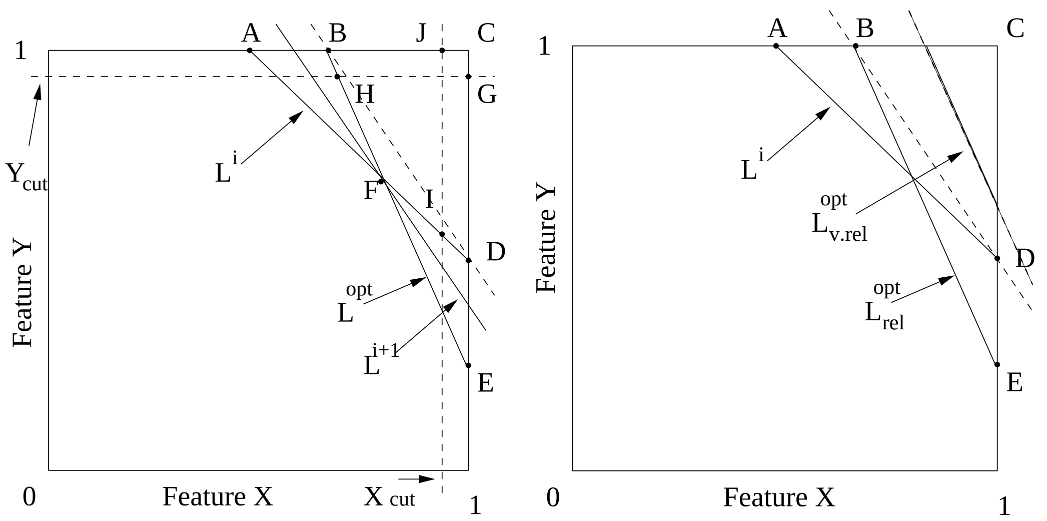
Chapter 4 discusses how the scores of the similarity predicates in the query condition are combined in the presence of weights. Modifying weights is one of the major mechanisms to improve the ranking of results. The goal of the query re-weighting techniques is to find the optimal relative weights among different components of the system. Formally, let $p_1, p_2, ..., p_n$ be the similarity predicates used to compute the corresponding similarity scores $s_1, s_2, ..., s_n$, and $w_1^{opt}, w_2^{opt}, \ldots, w_n^{opt}$ be the weights such that $\sum_{i=1}^{n} w_i^{opt} s_i$ captures the user's notion of similarity of a tuple to the users query. Let $w_1^0, w_2^0, \ldots, w_n^0$ be the weights initially associated with these predicates (for simplicity the system starts with equal weights for all predicates). Re-weighting modifies the weights associated with the predicates based on the user's feedback. The weights after the i-th iteration of the feedback process are denoted by $w_1^i, w_2^i, \ldots, w_n^i$. Re-weighting converges the weights associated with predicates to the optimal weights: $\lim_{i \longrightarrow \infty} v_j^i = v_j^{opt}$. For each predicate, the new weight is computed based on the similarity score of all attribute values for which relevance judgments exist. In section 7.2.2 we described how a *Scores* table is derived from the *Answer* table and *Feedback* table values by computing the scores for those attribute values for which feedback judgments are present. Each score is computed by applying its corresponding predicate to the value.

To develop the intuition of the query re-weighting strategy used, let us restrict ourselves to a simple situation in which a similarity condition for a query Q consists of two predicates: X and Y. The approach described in this simple context can be generalized to multiple predicates in a straightforward fashion.

For a query consisting of two predicates X, Y, there exist weights w_x^{opt}, and w_y^{opt} such that $w_x^{opt} s_x + w_y^{opt} s_y$ captures the similarity between a tuple t and the query condition from the user's viewpoint. The user's notion of similarity can be visualized as a line L^{opt}, illustrated in Figure 7.6, defined by the following equation:

$$w_x^{opt} \times s_x + w_y^{opt} \times s_y = \delta \tag{7.1}$$

where δ is a threshold so that any object whose similarity is greater than δ (i.e., all tuples above the line L^{opt}) will be ranked as relevant by the user. Figure 7.6 illustrates the similarity space with respect to predicates X and Y. Similarity values for predicates X and Y are in the range [0,1] where 1 is the highest similarity and 0 is the lowest similarity. The slope of L^{opt} represents the



a) Cut-off b) Relevance Level

Figure 7.6: Similarity Space

relative importance of X and Y based on user judgment.

The corresponding model used by the system to compute similarity to the query at any given instance (i.e., the weights w_x and w_y) can also be visualized as a line in the similarity space. Let L^i represent the line after the i-th iteration of query refinement, where L^i is:

$$w_x^i \times s_x + w_y^i \times s_y = \delta_1 \tag{7.2}$$

Consider the line BD joining the intercepts made by L^{opt} and L^i with line $Y = 1$ and $X = 1$ respectively in Figure 7.6(a). It can be shown that the slope of BD, that is, $-\frac{CD}{BC}$, lies in between the slope of L^i and that of L^{opt}. As a result, if the weights during the i-th iteration are modified such that the slope of L^{i+1} is equal to the slope of BD, then eventually L^i will converge to L^{opt}. To compute the slope of BD, the system attempts to estimate the lengths of CD and BC.

Such a re-weighting strategy, however, requires the system to know the slope $-\frac{CD}{BC}$. Since the system does not know L^{opt} (and hence cannot determine the points B and D a priori), it attempts to estimate the value of the lengths CD, and BC in order to estimate the slope $-\frac{CD}{BC}$ using the feedback about the relevance of the objects from the user.

To understand how the strategy works, notice that during an iteration, the system retrieves objects above line L^i (that is, in $\triangle ACD$) as relevant to the query. Of these objects, the user will mark only those objects that also lie within $\triangle BCE$ as relevant. That is, all objects within the region $BCDF$ will be marked as relevant to the query by the user. Based on this observation, many different strategies to estimate the slope $\frac{CD}{BC}$ can be developed. Two such strategies are explored next.

7.3.3.1 Minimum Weight Strategy

Let RL_x and RL_y be the set of values marked relevant by the user for predicates X and Y. Let

$$\Delta X = \min_{v_j \in RL_x} \{s_x(v_j)\} \tag{7.3}$$

$$\Delta Y = \min_{v_j \in RL_y} \{s_y(v_j)\} \tag{7.4}$$

where $s_x(v_j)$ and $s_y(v_j)$ are the similarity scores between v_j and the corresponding similarity predicate. ΔX and ΔY are used to estimate the points B and D in figure 7.6(a). That is, point B corresponds to $(\Delta X, 1)$ and the point D corresponds to $(1, \Delta Y)$. As a result,

$$\text{slope of line } BD = -\left(\frac{1-\Delta Y}{1-\Delta X}\right) \tag{7.5}$$

Since the slope of the line L^{i+1} is equal to the slope of line BD and given the constraints of the summation of the weights $w_x^{i+1} + w_y^{i+1} = 1$, the weights at the $i+1$-th iteration are updated as follows:

$$w_x^{i+1} = \alpha \times w_x^i + \beta \times \frac{1-\Delta Y}{2-\Delta X-\Delta Y} \tag{7.6}$$

$$w_y^{i+1} = \alpha \times w_y^i + \beta \times \frac{1-\Delta X}{2-\Delta X-\Delta Y} \tag{7.7}$$

where α and β ($\alpha + \beta = 1$) control how aggressively we modify the query condition. Using the *Scores* table of figure 7.2, we compute the new weight for $P(b)$ as: $\Delta_b = \min_{i \in relevant(b)}(P(b_i)) = \min(0.8, 0.9, 0.8) = 0.8$, similarly, $\Delta_c = 0.9$. If we take $\alpha = 0$ and $\beta = 1$, the new weights are: $w_b^{i+1} = \frac{1-.9}{2-.9-.8} = \frac{1}{3}$, and $w_c^{i+1} = \frac{1-.8}{2-.9-.8} = \frac{2}{3}$. We ignore non-relevant judgments for this strategy.

7.3.3.2 Average Weight Strategy

This approach estimates the lengths of BC and CD by averaging the scores of relevant values with respect to individual predicates X and Y. Consider Figure 7.6(a), all points above lines X_{cut} and Y_{cut} are very similar to the query based on features X and Y respectively. Given that X_{cut} and Y_{cut} are introduced at equal similarity thresholds, this approach estimates that the scores of objects similar to the query on individual predicates X and Y that are also relevant to the overall query are proportional to the size of the regions $CDIJ$ and $BCGH$ respectively and are proportional to the lengths of CD and BC. As a result,

$$\text{slope of line } BD = -\frac{\frac{\sum_{v_j \in RL_x}\{s_x(v_j) \mid s_x(v_j) > X_{cut}\}}{|RL_x|}}{\frac{\sum_{v_j \in RL_y}\{s_y(v_j) \mid s_y(v_j) > Y_{cut}\}}{|RL_y|}} \tag{7.8}$$

The strategy further extends to accommodate multiple levels of relevance (e.g., highly relevant, very relevant, and relevant, see section 7.2) that the user may associate with relevant values. In such a case, the user's model can be visualized in the similarity space as a sequence of parallel lines each corresponding to a different level of relevance as shown in Figure 7.6(b). L^{opt}_{rel} and $L^{opt}_{v.rel}$ represent relevant and very relevant thresholds respectively. To incorporate the similarity scores of values with their corresponding relevance levels to estimate the slope, let:

$$rel_X = \frac{\sum_{v_j \in RL_x} \{relevance(v_j) \times s_x(v_j) \mid s_x(v_j) > X_{cut}\}}{\mid RL_x \mid} \tag{7.9}$$

$$rel_Y = \frac{\sum_{v_j \in RL_y} \{relevance(v_j) \times s_y(v_j) \mid s_y(v_j) > Y_{cut}\}}{\mid RL_x \mid} \tag{7.10}$$

where $relevance(v_j)$ is the relevance level.

$$\text{slope of line } BD = -\frac{rel_X}{rel_Y} \tag{7.11}$$

The weight update policy is:

$$w_x^{i+1} = \alpha \times w_x^i + \beta \times \frac{rel_X}{rel_X + rel_Y} \tag{7.12}$$

$$w_y^{i+1} = \alpha \times w_y^i + \beta \times \frac{rel_Y}{rel_X + rel_Y} \tag{7.13}$$

Using the data from figure 7.2 (and $\alpha = 0$, $\beta = 1$), we compute the new weight for $P(b)$ as: $rel_b = \frac{\sum_{b_i \in RL(b)} P(b_i)}{|RL_b|} = \frac{0.8+0.9+0.8-0.3}{4} = 0.55$, similarly, $rel_c = 0.9$. Then $w_b^{i+1} = \frac{.55}{.55+.9} = \frac{.55}{1.45} = 0.38$ and $w_b^{i+1} = \frac{.9}{.55+.9} = \frac{.9}{1.45} = 0.62$.

In both strategies, if there are no relevance judgments for any tuples involving a predicate, then the original weight is preserved as the new weight. These strategies also apply to predicates used as a join condition such as shown in the example of figure 7.3.

7.4 Intra-Predicate Query Refinement

In intra-predicate refinement, the similarity predicates are modified in several ways. Predicates, in addition to a single query point (value), may use a number of points as a query region, possibly with its own private, attribute-specific similarity aggregation function [125]. For example, multiple points are aggregated with a function λ (figure 7.1) with weights for each query point.

Intra-predicate refinement is by its very nature *domain dependent*, hence each user-defined type specifies its own *scoring* and *refinement* functions to operate on objects of that type. We present some strategies in the context of example 3.2.1 and describe how they apply to the $close\_to\_refine$ function. As described above, the parameters to this function are a list of values and corresponding relevance judgments: $close\_to\_refine(list\langle location\_value, relevance\rangle)$. In the context of figure 7.2, let attribute b be of type *location*, and P be the *close_to* function, then we invoke $close\_to\_refine($

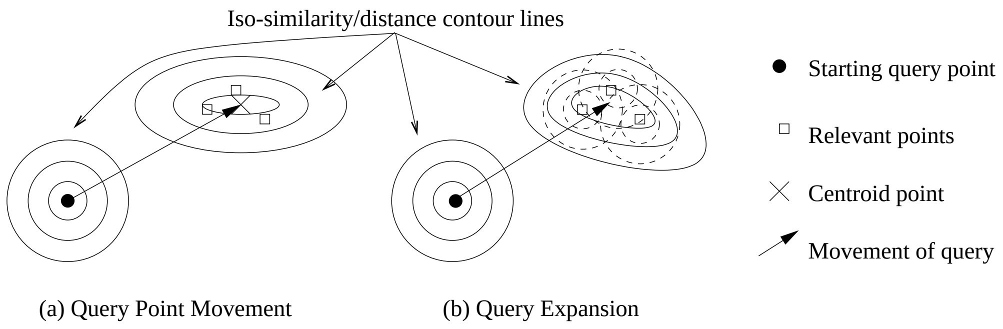


Figure 7.7: Intra-predicate Refinement

$\langle b_1, 1\rangle$, $\langle b_2, 1\rangle$, $\langle b_3, 1\rangle$, $\langle b_4, -1\rangle$), since those are the tuples the user gave feedback on for attribute b. Figure 7.1 illustrates the following approaches for this type:[2]

7.4.1 Distance Metric Selection

Distance metric selection (DMS) refers to determining which of a set of distance metrics is most consistent with the supplied feedback. For example, any L_p metric can be used, not just Euclidean distance (L2). The *close_to* function uses Euclidean distance (L2), but Manhattan distance (L1) can also be used for geographic locations. We compute the L1 and L2 based distances between all relevant values (b_1, b_2, b_3) and average them. The metric that gives the least average distance is chosen as the distance metric, since it results in the highest scores, on average, for the relevant values.[3] Other distance metrics may be similarly tested for fit.

7.4.2 Query Re-weighting

Query re-weighting assigns a weight to each dimension of the query vector. The weight assigned is inversely proportional to the standard deviation of feature values of the relevant objects along that dimension. Intuitively, among the relevant objects, the higher the variance along a dimension, the lower the significance of that dimension [78, 134]. For example, if the distance function is Euclidean, the function changes from (7.14) to (7.15):

$$\text{OldDistance}(C, P) = \sum_{j=1}^{m} (C[j] - P[j])^2 \tag{7.14}$$

$$\text{NewDistance}(C, P) = \sum_{j=1}^{m} \frac{1}{\sigma_j} (C[j] - P[j])^2 \tag{7.15}$$

where σ_j is the standard deviation of feature values of the relevant objects along dimension j.

[2]Note that this discussion uses distance functions rather than similarity functions since it is more natural to visualize the distance between points than their similarity; distance is converted to similarity as presented in example 3.2.1.

[3]Although this algorithm is $O(n^2)$, in practice there are very few (<20) feedback values simplifying this problem.

Figure 7.7 (a) shows how the distance function under re-weighting changes its shape. The figure shows contours representing equidistant ranges from the new query which have now become ovals rather than circles.

Following our example, if $b_1.x$, $b_2.x$, and $b_3.x$ are similar (i.e., small variance), and $b_1.y$, $b_2.y$, and $b_3.y$ are different (i.e., larger variance), we infer that the x dimension captures the users intention better, and set $w_x = \frac{1}{std\ dev(b_x)}$ and $w_y = \frac{1}{std\ dev(b_y)}$, and then re-normalize the weights such that $w_x + w_y = 1$.

7.4.3 Query Point Movement

The Query point movement (QPM) approach is based on Rocchios method for vector space models [130, 134], it uses a single multidimensional object per domain as the query value. When the user uses multiple examples to construct the query, the centroid is used as the single point query. Similarly, at each iteration of query refinement, a new centroid is formed by combining the old query value (centroid) with the weighted centroid of the values the used marked relevant and non-relevant. The weights are obtained from the level of relevance provided by the user. Let $v_1, v_2, \ldots v_n$ denote n vectors marked relevant by the user and $u_1, u_2, \ldots u_m$ denote m vectors marked non-relevant by the user, and let r_{v_i} and r_{u_i} be their corresponding levels of relevance. Let $p[j]$ denote the value of point p along the j-th dimension of the space, $1 \leq j \leq l$, l being the dimensionality of the space. The new weighted centroid C_{new} is defined as:

$$C_{new}[j] = \alpha \times C_{old}[j] + \beta \times \frac{\sum_{i=1}^{n} r_{v_i} \times v_i[j]}{\sum_{i=1}^{n} r_{v_i}} - \gamma \times \frac{\sum_{i=1}^{m} r_{u_i} \times u_i[j]}{\sum_{i=1}^{n} r_{u_i}} \tag{7.16}$$

Where C_{old} is the old centroid. The constants α, β, and γ regulate the speed at which the query point moves towards relevant values and away from non-relevant values. By moving the query point closer to relevant values and away from non-relevant values, it better captures the users intentions.

Figure 7.7 (a) shows how the query point changes location in the query point movement approach. The figure shows contours representing equidistant ranges from the new query.

In our example, the query value $\hat{b}$ migrates to $\hat{b}$' by: $\hat{b}' = \alpha \times \hat{b} + \beta \times \frac{\sum(b_1,b_2,b_3)}{3} - \gamma \times \frac{\sum(b_4)}{1}$, $\alpha + \beta + \gamma = 1$

7.4.4 Query Expansion

Unlike the query point movement approach, the query expansion (QEX) approach uses multiple objects per attribute as a *multi-point query* [125, 169]. When the user marks several points as relevant, the query processor selects a small number of good representative points to construct the multi-point query by clustering the set of relevant points and using the centroids of the clusters as the representatives. Possible algorithms to cluster the points include the doubling algorithm [31], k-means algorithm, or even the technique presented in FALCON [169].

The representative points are used to construct the new multi-point query. Notice that the query from the previous iteration does not directly affect the new query. But in constructing the query, objects deemed relevant during previous iterations are also incorporated into the clusters. Implicitly, relevant points get added while the non-relevant ones get dropped as the refinement process moves from one iteration to the next.

The weight for each cluster centroid in this approach is proportional to the number of relevant objects in the corresponding cluster. The weights are added to the multi-point query along with the corresponding representatives. The distance from the multi-point query is defined as the weighted combination of the individual distances from the representatives. Figure 7.7 (b) shows the distance function for multi-point queries. The dashed lines are contours representing equidistant ranges for each of the representatives while the solid lines are contours representing equidistant ranges from the entire multi-point query.

7.4.5 Similarity Join Considerations

Predicates used as similarity join conditions do not have constant query points, therefore the techniques we described that rely on query points do not apply to join predicates, but the remaining still do. Intra-predicate *query point movement* and *query expansion* rely on a known set of query values and therefore do not apply, but *query re-weighting* and *distance metric selection* still apply.

7.5 Evaluation

Similarity retrieval and query refinement represent a significant departure from the existing access paradigm (based on precise SQL semantics) supported by current databases. Our purpose in this section is to show some experimental results, and illustrate that the proposed access mechanism can benefit realistic database application domains. We study our systems various components, and present a realistic sample e-commerce catalog search application to validate the approach.

7.5.1 Methodology

To evaluate the quality of retrieval we establish a baseline ground truth set of relevant tuples. This ground truth can be defined objectively, or subjectively by a human users perception. When defined subjectively, the ground truth captures the users information need and links the human perception into the query answering loop. In our experiments we establish a ground truth and then measure the retrieval quality. We measure the quality of retrieval with precision and recall [137]. *Precision* is the ratio of the number of relevant tuples retrieved to the total number of retrieved tuples: $precision = \frac{|relevant \bigcap retrieved|}{|retrieved|}$, while *recall* is the ratio of the number of relevant tuples retrieved to the total number of relevant tuples: $recall = \frac{|relevant \bigcap retrieved|}{|relevant|}$. We compute precision and recall after each tuple is returned by our system in rank order.

The effectiveness of refinement is tied to the convergence of the results to the users information need. Careful studies of convergence have been done in the past in restricted domains, e.g., Rocchio [130] for text retrieval, MARS [134, 125], Mindreader [78], and FeedbackBypass [11] for image retrieval, and FALCON [169] for metric functions. We believe (and our experiments will show) that these convergence experiments carry over to the more general SQL context in which we are exploring refinement.

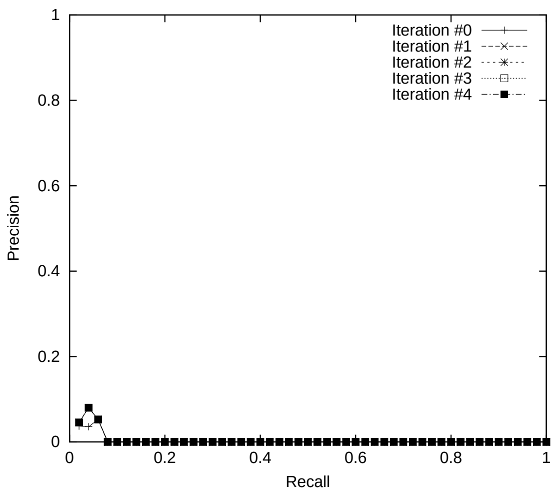
7.5.2 Numeric Dataset Experiment

For this experiment, we used two datasets. One is the fixed source air pollution dataset from the AIRS[4] system of the EPA and contains 51,801 tuples with geographic location and emissions of 7 pollutants (carbon monoxide, nitrogen oxide, particulate less than 2.5 and 10 micrometers, sulfur dioxide, ammonia, and volatile organic compounds). The second dataset is the US census data with geographic location at the granularity of one zip code, population, average and median household income, and contains 29470 tuples. We used several similarity predicates and refinement algorithms.

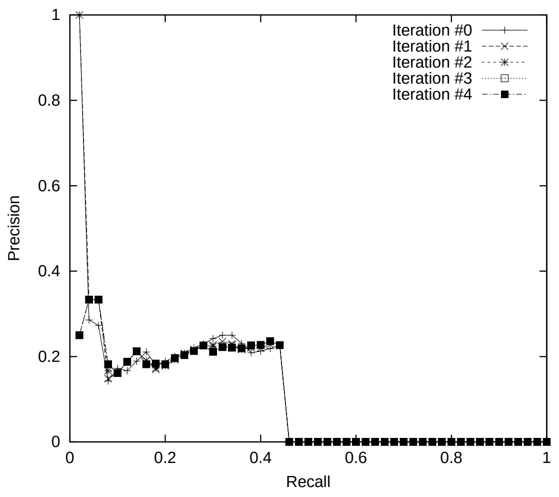
For the first experiment we implemented a similarity predicate and refinement algorithm based on FALCON [169] for geographic locations, and a query point movement and dimension re-weighting for the pollution vector of the EPA dataset. We started with a conceptual query looking for a specific pollution profile in the state of Florida. We executed the desired query and noted the first 50 tuple as the ground truth. Next, we formulated this query in 5 different ways, similar to what a user would do, retrieved only the top 100 tuples, and submitted tuple level feedback for those retrieved tuples that are also in the ground truth. Here we performed 5 iterations of refinement. Figure 7.8a) shows the results of using only the location based predicate, without allowing predicate addition. Similarly figure 7.8b) shows the result of using only the pollution profile, without allowing predicate addition. Note that in these queries feedback was of little use in spite of several feedback iterations. In figure 7.8c) we instead used both predicates but with the default weights and parameters, notice how the query slowly improves. Figure 7.8d) starts the query with the pollution profile only, but allowing predicate addition. The predicate on location is added after the first iteration resulting in much better results. Similarly figure 7.8e) starts only with the location predicate. The initial query execution yields very low results, but the pollution predicate is added after the initial query resulting in a marked improvement. In the next iteration, the scoring rule better adapts to the intended query which results in another high jump in retrieval quality. It is sufficient to provide feedback on only a few tuples, e.g., in this query, only 3 tuples were submitted for feedback after the initial query, and 14 after the first iteration. The number of tuples with feedback was similarly low (5%-20%) in the other queries.

In the second experiment, we use a join query over the EPA and census datasets. Notice that we cannot use the location similarity predicate from the first experiment since the FALCON [169]

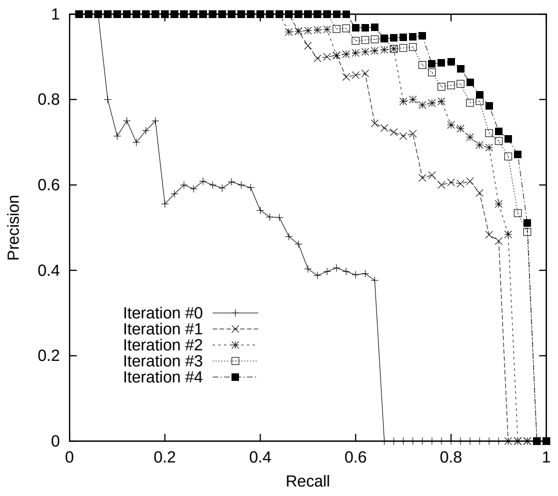
[4]http://www.epa.gov/airs



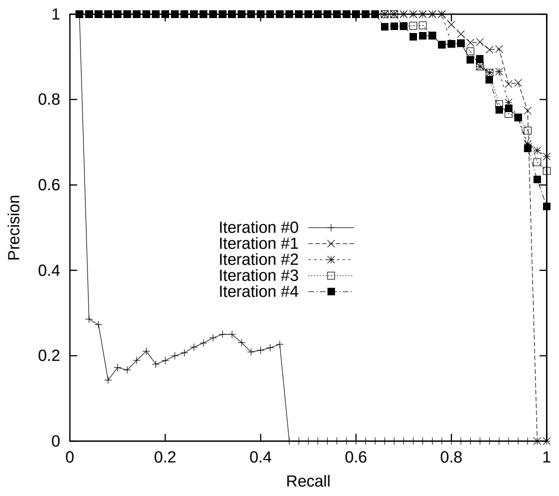
a) Location alone



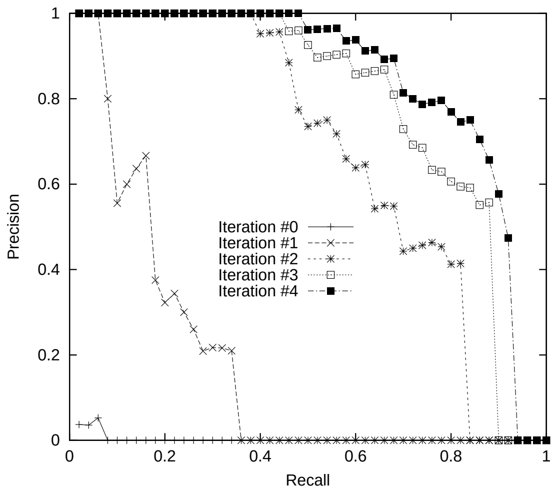
b) Pollution alone



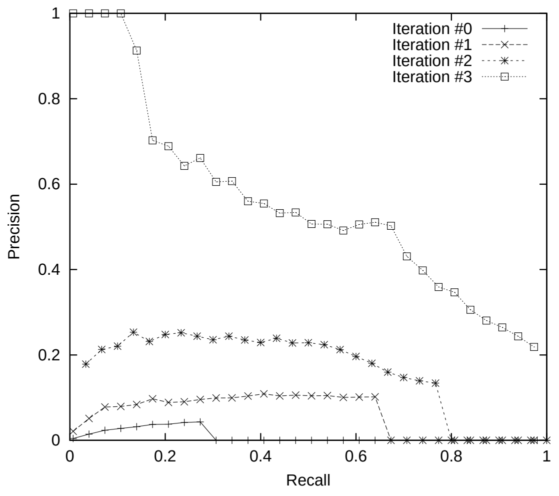
c) Location and pollution



d) Pollution, add location pred.



e) Location, add pollution pred.



f) Similarity join query

Figure 7.8: Precision–recall Graphs for several Iterations, various Refinement Strategies

based similarity predicate is not joinable. This is because it relies on a set of "good points" which must remain the same over the execution of a query iteration. If we change the set of good points to a single point from the joining table in each call, then this measure degenerates to simple Euclidean distance and the refinement algorithm does nor work. Figure 7.8f) shows the results of a query where the census and EPA datasets are joined by location, and we're interested in a pollution level of 500 tons per year of particles 10 micrometers or smaller in areas with average household income of around $50,000. We constructed the ground truth with a query that expressed this desire and then started from default parameters and query values.

7.5.3 Multidimensional Dataset Experiment

For this experiment, we use the Corel image feature collection.[5] The collection consists of 68,040 images with a brief textual description for each image. We extracted color (4×8 color histogram) and texture (16-dimensional co-occurrence texture) features (see appendix A) for each image and used a text vector model [137] for its text description. Each extracted feature corresponds to an attribute of the relation storing the Corel images. For a given query type (e.g., a color or a color-texture query), we select an example query Q randomly from the dataset and retrieve the top 50 answers. The answers are computed based on fixed intra-predicate similarity functions and a random choice of predicate weights (for multi-predicate queries).[6] We refer to this set as the relevant set $relevant(Q)$. We construct the starting query by slightly disturbing Q (i.e., by choosing a query point close to Q), setting equal predicate weights and retrieving the top 100 answers which become the retrieved set $retrieved(Q)$. All tuples in $retrieved(Q)$ that are also in $relevant(Q)$ are submitted as relevant feedback to the system, which refines the query based on the feedback, evaluates the refined query, and returns the new answers. This process is repeated over several iterations. Precision and recall are computed using $relevant(Q)$ and $retrieved(Q)$ as discussed above, all measurements are averaged over 20 queries.

We first demonstrate the effectiveness of the intra-predicate strategies discussed in section 7.4 by executing a single predicate query (a color-similarity query, i.e., request for images similar to a given image in terms of color). We use the L1 metric as the distance function (equivalent to the histogram intersection similarity metric [115]), and query expansion for query point selection. Figure 7.9 shows that the quality of results improves from one iteration to the next (i.e., $retrieved(Q)$ approaches $relevant(Q)$) as the system learns the query points and the dimensional weights that capture the users information need. We next evaluate the inter-predicate reweighting techniques by executing a multi-predicate conjunction query (a color-and-texture query, i.e., request for images similar to a given image in terms of both color and texture). We use the minimum weight strategy for inter-predicate re-weighting, and query expansion and re-weighting for intra-predicate refinement. Figure 7.10 shows that inter-predicate reweighting, in conjunction with intra-predicate refinement techniques, improves the retrieval effectiveness of multi-predicate queries significantly.

[5]Available online at http://kdd.ics.uci.edu.

[6]The weights chosen capture the user perception for the query; a goal of query refinement is to learn those weights.

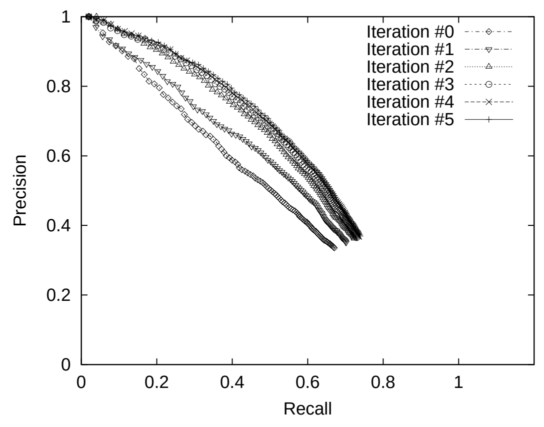


Figure 7.9: Retrieval Performance of Single-predicate Query

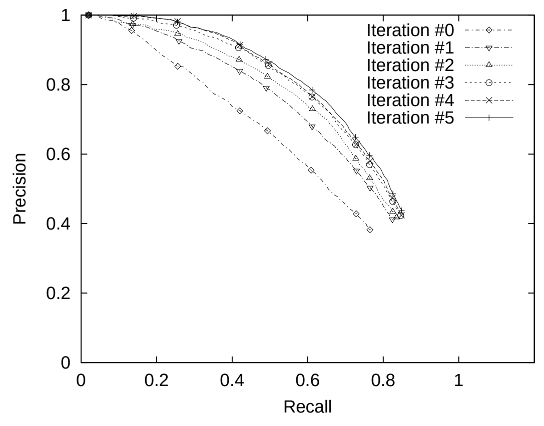


Figure 7.10: Retrieval Performance of Multi-predicate Query using Intra-predicate Query Expansion and Inter-predicate Reweighting

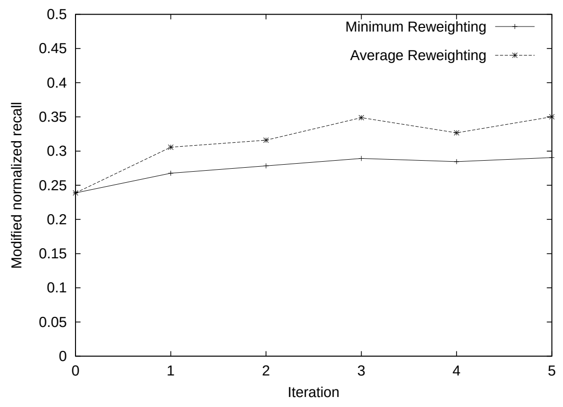


Figure 7.11: Comparison of the two Inter-predicate Weight Selection Techniques

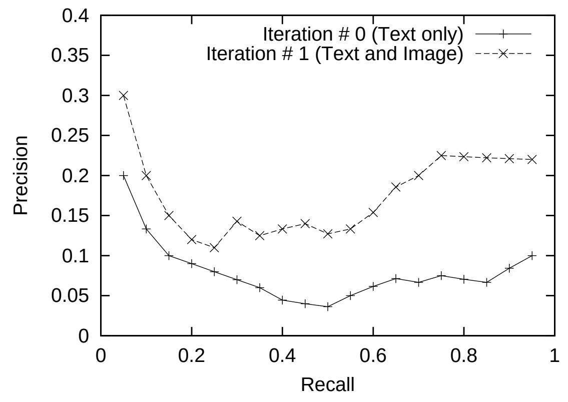


Figure 7.12: Retrieval Performance of Multi-predicate Query with Predicate Addition

Figure 7.11 compares the two inter-predicate weight selection techniques for the same query (color-and-texture query). In figure 7.11 we compare the ground truth and ranked answer lists with the *normalized recall* metric[7] discussed in appendix B to evaluate the *inter-predicate* re-weighting strategies of section 7.3. Figure 7.11 shows how refinement converges, the 4th iteration shows some over-fitting which is corrected in the 5th iteration. Figure 7.12 shows the effect of predicate addition to a query. The starting query consists of only a text predicate (e.g., description *similar-to* some_description(s)). Based on the feedback, the system decides that the users information need also involves visual similarity and adds an image predicate (i.e., color *similar-to* some_image(s) *and* texture *similar-to* some_image(s)) to the query, resulting in increased retrieval effectiveness.

7.5.4 Sample E-Commerce Application of Similarity Retrieval and Refinement

To illustrate a realistic application, we built a sample E-commerce application on top of our query refinement system, to explore the utility of our model in this context. This application is described in detail in chapter 9. We explore how the granularity (i.e., tuple vs. column level feedback) and the amount of feedback affect the quality of results.

7.5.4.1 Example Queries

We explore our system through the following conceptual query: "men's red jacket at around $150.00". We browsed the entire collection and constructed a set of relevant results (ground truth) for the query, we found 10 items out of 1747 to be relevant and included them in the ground truth. Several ways exist to express this query, we try the following:

1. Free text search of *type*, *short* and *long description* for "men's red jacket at around $150.00".
2. Free text search of *type* with "red jacket at around $150.00" and *gender* as male.
3. Free text search of *type* with "red jacket", *gender* as male, and *price* around $150.00.
4. Free text search of *type* with "red jacket", *gender* as male, *price* around $150.00, and pick a picture of a red jacket which will include the image features for search.

7.5.4.2 Evaluation

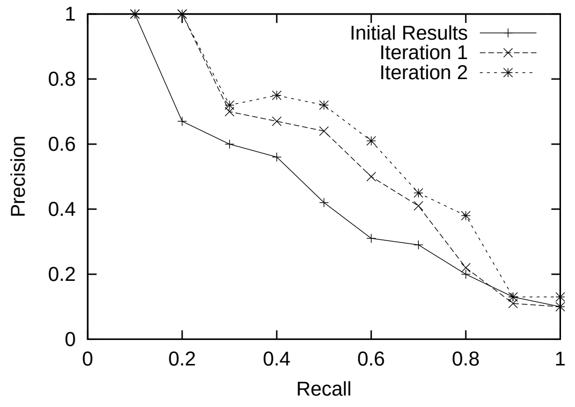
We ran all four queries with feedback to obtain a set of precision/recall values which are shown in table 7.1. The table shows precision values for several recall percentiles. We show one refinement iteration for queries 1 and 2, and two iterations for queries 3 and 4. The refined answers are an improvement of the original results in general, and specifically for Query 4, as can be seen in table 7.1.

Effect of Feedback Granularity on Refinement Quality. Our framework provides for two granularities of feedback: tuple and column level. In tuple level feedback the user judges the tuple as a whole, while in column level she judges individual attributes of a tuple. Column level feedback presents a higher burden on the user, but can result in better refinement quality. This is shown in

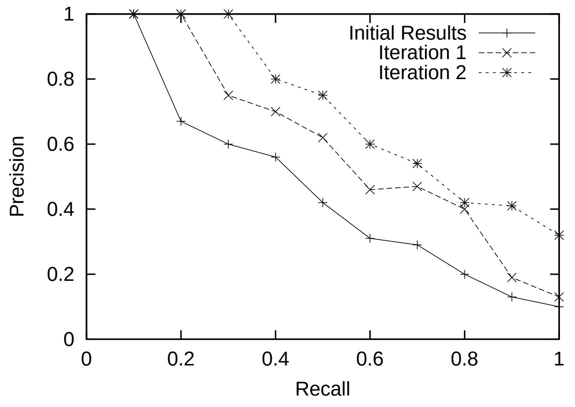
[7] *Normalized recall* measures the similarity between two ranked lists, it returns a similarity value in the range [0,1].

Recall	Query 1 precision		Query 2 precision		Query 3 precision			Query 4 precision		
	initial results	1st ref	initial results	1st ref.	initial results	1st ref.	2nd ref.	initial results	1st ref.	2nd ref.
10%	0.07	1	0.25	1	0.17	1	1	1	1	1
20%	0.11	0.15	0.33	1	0.29	1	1	0.6	1	1
30%	0.15	0.19	0.38	1	0.20	0.75	1	0.75	1	1
40%	0.13	0.19	0.44	0.80	0.25	0.57	1	0.44	1	1
50%	0.15	0.21	0.36	0.45	0.26	0.56	1	0.42	0.71	1
60%	0.13	0.21	0.21	0.22	0.25	0.55	0.67	0.43	0.60	0.75
70%	0.11	0.13	0.18	0.20	0.25	0.47	0.64	0.47	0.64	0.78
80%	0.12	0.09	0.18	0.17	0.24	0.32	0.35	0.5	0.57	0.5
90%				0.15	0.17	0.20	0.35	0.38	0.39	0.53
100%					0.11	0.17	0.3	0.23	0.23	0.29

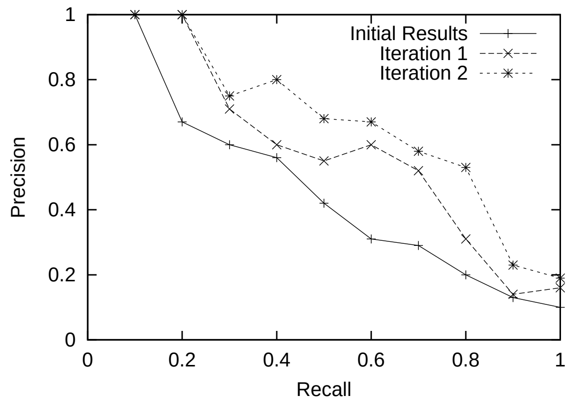
Table 7.1: Comparison of Different Queries and their Refinement



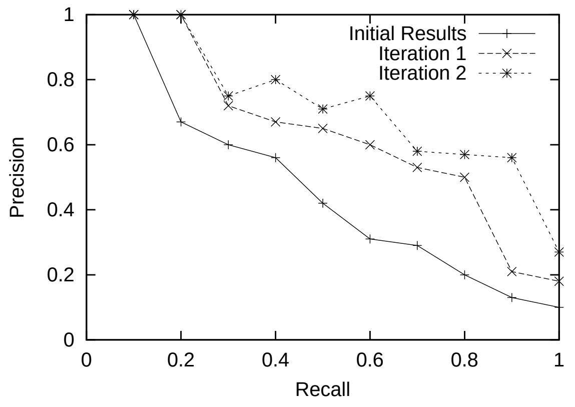
a) Tuple level Feedback with feedback on 2 Tuples



b) Column level Feedback with feedback on 2 Tuples



c) Tuple level Feedback with feedback on 4 Tuples



d) Tuple level Feedback with feedback on 8 Tuples

Figure 7.13: Precision–recall Graphs averaged for 5 Queries for different amount and granularity of Feedback

figures 7.13a) which shows tuple level feedback and 7.13b) which shows column level feedback. For these figures we used 5 different queries with their ground truth, gave feedback on exactly 2 tuples and averaged the results. For tuple level feedback, 2 entire tuples were selected, for column level feedback, we chose only the relevant attributes within the tuples and judged those. As expected, column level feedback produces better results.

Effect of Amount of Feedback Judgments on Refinement Quality. While in general the refinement system adapts better to the users information need when it receives more relevance judgments, in practice this is burdensome to users. The amount of feedback may vary from user to user; here we explore how the amount of feedback affects the retrieval quality.

Empirically, even a few feedback judgments can improve query results substantially. We explore the same 5 queries from above and give tuple level feedback for 2, 4 and 8 tuples. The results are shown in figures 7.13a,c,d) respectively. More feedback helps improve the results, but with diminishing returns, in fact column level feedback for 2 tuples is competitive with tuple level feedback for 8 tuples. Based on these observations, we conclude that users need not expend a huge effort in feedback to improve their answers.

7.6 Related Work

While there is much research on improving the effectiveness of queries in specific applications, we are not aware of work that addresses generalized query refinement in general purpose SQL databases. The work most related to our research is relevance feedback and query refinement in multimedia and Information Retrieval (IR) systems, and cooperative databases or query formulation tools.

Traditionally, similarity retrieval and relevance feedback have been studied for textual data in the IR community and have recently been generalized to other domains. IR has developed numerous models for the interpretation of similarity [137, 8], e.g., Boolean, vector, probabilistic, etc. models. IR models have been generalized to multimedia documents, e.g., image retrieval [134, 125, 115, 52] uses image features to capture aspects of the image content, and adapts IR techniques to work on them.

Query refinement through relevance feedback has been studied extensively in the IR literature [137, 8, 130] and has recently been explored in multimedia domains, e.g., for image retrieval by Mindreader [78], MARS [134, 125], and Photobook [103], and for time-series retrieval [82], among others. FALCON [169] uses a multi-point example approach similar to our multi-point approach and generalizes it to any suitably defined metric distance function, regardless of the native space. FALCON assumes that all examples are drawn from the same domain and does not handle multiple independent attributes. Under our framework, we consider FALCON as a non-joinable algorithm since it is dependent on the good set of points remaining constant during a query iteration. FALCON however fits perfectly in our framework as an intra-predicate algorithm for selection queries as we discussed in section 7.4 and showed in our experiments. In [98] the authors implement query refinement by layering a text vector model IR system with relevance feedback on top of a SQL database; our work is different since we support arbitrary data types and refinement across them. Given the applicability of refinement to textual and multimedia data, our work takes the next step, extending refinement to general SQL queries where user-defined functions implement similarity and refinement models for data-types such as text, image, audio, time-series, etc. [133] focuses on adjusting relative weights among different features. The model for reweighting used is similar to our counting approach. MindReader [78] also applied a query modification technique which is similar to our intra-predicate re-weighting approach. It focuses on a single multidimensional space, and its goal is to find a single representative point that minimizes the stress based on the Euclidean distance function. A similar technique has been used by Hull [76] (Local LSI). Hull attempts to find the correlation among relevant vectors by applying singular value decomposition (SVD) to relevant vectors and creating a new relevant space. Then all or a subset of vectors are mapped into this new space. New vectors that fit in the new space are considered to be relevant. A disadvantage of this approach is that SVD is an extremely expensive computation. The difference between Local LSI and MindReader is that Local LSI is applied to text document collections which are highly multi-dimensional while MindReader is targeted to two-dimensional map data. In [37], Cox et. al. use a different strategy to learn user's intention. They focus on a probabilistic model to learn from the user response. The system does not return back an answer set to the user after each iteration of the user response. Instead, the system poses a question to the user. The question is supposed to be the most discriminating criteria so that the number of questions needed to be asked is minimum. This approach is useful for a user to find a starting sample to feed to another retrieval system. But if the user already has sample data that he can directly feed to other systems, the approach imposes a burden on the user to go through the iteration. MEGA follows the same approach utilizing feedback at each iteration to determine which dimensions in a vector are the most discriminating. At each iteration, it returns results that will provide it the best discriminatory power when feedback is provided, these are not necessarily the best results for the query.

Usability and usefulness of query refinement is of great importance to our work. Relevance feedback and query refinement are generally highly praised in the IR community [137, 8]. Despite many advances, a recent study [80] on its use in web search engines suggests that users do not rely heavily on relevance feedback tools during search. Users reformulate their queries manually about twice as frequently as by using provided feedback tools. Two explanations are offered, (1) manual query reformulation is easy for textual queries and therefore does not benefit much from relevance feedback, and (2) some evidence points towards a usability problem unrelated to the technology itself. We distinguish ourselves from this perspective since our work is independent of any specific user interface, and manual refinement of a query with complex attributes (e.g., visual, geographic, etc.) is significantly more difficult than a textual query, thus benefiting from our approach.

Cooperative databases seek to assist the user to find the desired information through interaction. VAGUE [106], in addition to adding distance functions to relational queries, helps the user pose imprecise queries. While VAGUE does not support query refinement, in a sense it out-sources the *decision* stage of refinement to the user, it examines a user query and for each distance (similarity) predicate, asks the user for the correct interpretation, including weights, and then proceeds to execute the query. A related interface, FLEX [107] refines precise queries from a vague description to a specific query suitable for execution. The goal is not to refine the query based on the value of answers, but instead to formulate queries and resolve conflicts regarding the number of results. The Eureka [143] browser is a user interface that lets users interactively manipulate the conditions of a query with instantly update the results. The goal of the CoBase [34] cooperative database system is to avoid empty answers to queries. It uses Type Abstraction Hierarchies (TAHs) and relaxation operators to relax query conditions and expand the answer set. CoBase is different from our approach: relaxation operators influence existing query conditions but cannot add new ones or change their interpretation based on the value of query results, relaxation via TAHs roughly corresponds to decreasing the similarity threshold in our framework and using query point movement. In [83], the authors consider a succession of *manually* modified *precise* queries to be a *browsing session* (session query) and focus on optimizing the computation of results. Agrawal [2] proposes a framework where users submit preferences (feedback) and explains how they may be composed into compound preferences.

7.7 Conclusions

The Intra- and Inter-predicate strategies discussed effectively span the whole *range of action* in our design space, from single predicates to balancing multiple predicates, to adding new predicates to the query. Adding predicates also addresses cases where the initial predicates are too few and the query needs to be focused and tightened. Finally we used both tuple level and attribute level feedback and showed how they can coexist. A practical system implementation may implement all or some of these techniques, such an implementation can then be classified using our design space.

Chapter 8

Query Refinement Processing

8.1 Introduction

Top-k selection queries are becoming common in many modern-day database applications. Unlike in a traditional relational database system (RDBMS) where a selection or join query consists of a precise condition and the user expects to get back the exact set of objects that satisfy the condition, in top-k queries, the user specifies a similarity or distance criteria that may include target values for certain attributes, and does not expect exact matches to these values in return. Instead, the result to such queries is typically a *ranked list* of the "top k" objects that best match the given attribute values.

An important aspect of top-k queries is user subjectivity [106, 45]. To return "good quality" answers, the system must understand the user's *perception* of similarity, i.e., the relative importance of the attributes/features[1] to the user. The system models user perception via the distance functions (e.g., Euclidean in the above example) and the weights associated with the features [46, 27, 122, 33]. At the time of the query, the system *acquires* information from the user based on which it determines the weights and distance functions that best capture the perception of this particular user and instantiates the model with these values. Note that this instantiation is done at query time since the user perception differs from user to user and from query to query. Once the model is instantiated, we retrieve, based on the model, the top answers by first executing a k nearest neighbor (k-NN) algorithm on each individual feature[2] and then *merging* them to get the overall answers [115, 109, 46, 45].

As discussed in chapters 6 and 7, due to the subjective nature of top-k queries, the answers returned by the system to a user query usually do not satisfy the user's need right away. In this case, the user would like to *refine* the query and resubmit it in order to get back a better set of answers.

While there has been a lot of research on improving the effectiveness of query refinement as well

[1]We use the terms attribute and feature interchangeably in this chapter.

[2]In this chapter, we assume that all the feature spaces are metric and an index (called the Feature-index or F-index) exists on each feature space. A F-index is either single dimensional (e.g., B-tree) or multidimensional (e.g., R-tree) depending on the feature space dimensionality.

as on evaluating top-k queries efficiently, there exists no work that we are aware of on *how to support refinement of top-k queries efficiently inside a DBMS*. We explore such approaches in this chapter. A naive approach to supporting query refinement is to treat a refined query just like a starting query and execute it from scratch. We observe that the *refined queries are not modified drastically* from one iteration to another; hence executing them from scratch every time is wasteful. Most of the execution cost of a refined query can be saved by appropriately *exploiting the information generated during the previous iterations of the query*. We show how to execute subsequent iterations of the query efficiently by utilizing the cached information. Note that since the query changes, albeit slightly, from iteration to iteration (i.e., the query points and/or the weights/distance functions are modified), we, in general, cannot answer a refined query entirely from the cache, i.e., we still need to access some data from the disk. Our technique minimizes the amount of data that is accessed from disk to answer a refined query. Furthermore, our technique does not need to examine the entire cached priority queue to answer a refined query, i.e., it explores only those items in the cache that are necessary to answer the query, thus saving unnecessary distance computations (CPU cost).[3] Our experiments on real-life datasets show that our techniques improve the execution cost of the refined queries by up to two orders of magnitude over the naive approach.

A secondary contribution of this chapter is a technique to evaluate *multipoint queries* efficiently. In order to support refinement, we need to be able to handle multipoint queries as shown in the MARS and FALCON papers [122, 27, 169]. Such queries arise when the user submits multiple examples during feedback. We first formally define the multipoint query and then develop an efficient k-NN algorithm that computes the k nearest neighbors to such queries. Our experiments show that our algorithm is more efficient compared to the multiple expansion approach proposed in FALCON [169] and MARS [125].

8.2 The Model

In top-k selections and joins, the user poses a query Q by providing a similarity criteria between values (similarity function), target query values (for selection attributes), and by specifying the number k of matches desired. The target values can be either explicitly specified by the user or are extracted from a user supplied example. We refer to Q as the 'starting' query. The starting query is then executed and the top k matches are (incrementally) returned. If the user is not satisfied with the answers, she provides feedback to the system either by submitting interesting examples or via explicit weight modification. Based on the feedback, the system refines the query representation to better suit the user's information need. The 'refined' query is then evaluated and the process continues for several iterations until the user is fully satisfied. When the user is satisfied with the answers returned, she can request for additional matches incrementally. The process of feedback and requesting additional matches can be interleaved arbitrarily. We now discuss how each object

[3]Our techniques are independent of the way the user provides feedback to the system, i.e., it does not matter whether she uses the QBE (i.e., "give me more like this") interface or the explicit weight modification interface.

Symbol	Definition	Symbol	Definition
S	Multidimensional space	d_S	Dimensionality of S
O	A database object/record	Q	User query (multipoint)
n_Q	Number of points in Q	$\mathcal{P}_Q$	Set of n_Q points in Q
$P_Q^{(i)}$	The ith point in $\mathcal{P}_Q$	$\mathcal{W}_Q$	Set of n_Q weights associated with $\mathcal{P}_Q$
$w_Q^{(i)}$	ith weight in$\mathcal{W}_Q$ (associated with $P_Q^{(i)}$)	${\mu_Q}^{(j)}$	Weight (intra-feature) for jth dimension of S
$\mathcal{D}_Q$	Distance function for Q	$\mathcal{D}_Q(Q,O)$	Overall distance between Q and O
QPM	Query Point Movement	QEX	Query Expansion
FR	Full Reconstruction Approach	SR	Selective Reconstruction Approach
LBD	Lower Bounding Distance	BR	Bounding Rectangle

Table 8.1: Summary of Symbols, Definitions and Acronyms

is represented in the database (the object model), how the user query is represented (the query model), how the retrieval takes place (the retrieval model) and how refinement takes place (the refinement model).

8.2.1 Object Representation

Objects are vectors in a multidimensional space. Let S be a d_S dimensional space, we view an object O as a point in this multidimensional space, i.e., O is a d_S dimensional vector. Many image retrieval systems represent image features in this way. How the objects O are obtained (i.e., the feature extraction) depends on the application (e.g., in example 1.0.1, special image processing routines are used to extract the color and texture from the image).

8.2.2 Query Representation

A query is represented as a distance function $\mathcal{D}_Q$ that computes the distance between any object and a set $\mathcal{P}_Q$ of query objects, all from the multidimensional space S. For selection queries, $\mathcal{P}_Q$ is a set of query objects supplied by the user, or manipulated via query refinement. For join queries, $\mathcal{P}_Q$ iterates through the values of one of the relations being joined, one value at a time. The distance function $\mathcal{D}_Q$ associates weights μ_Q with each dimension of the space S, and a weight w_Q with each query point. The reason for allowing multiple points in the query is that for selection queries, during refinement, the user might submit multiple examples to the system as feedback (those that she considers relevant) leading to multiple points in the space (cf. section 8.2.3). We refer to such queries as *multipoint queries*. The user may also specify the relative importance of the submitted examples, i.e., the importance of each example in capturing her information need (e.g., relevance levels in MARS [132], "goodness scores" in Mindreader [78]). To account for importance, we associate a weight with each point of the multipoint query. Note that allowing multiple query points does not affect the interpretation of joins where we substitute one at a time, the values from one of the relations as a query point. We now formally define a multipoint query:

Definition 8.2.1 (Multipoint Query) *A multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q \rangle$ for the space S consists of the following information:*

- *The number n_Q of points in Q.*

- *A set of n_Q points $\mathcal{P}_Q = \{P_Q{}^{(1)}, ..., P_Q{}^{(n_Q)}\}$ in the d_S-dimensional feature space S.*

- *A set of n_Q weights $\mathcal{W}_Q = \{w_Q{}^{(1)}, ..., w_Q{}^{(n_Q)}\}$, the ith weight $w_Q{}^{(i)}$ being associated with the ith point $P_Q{}^{(i)}$ $(1 \geq w_Q{}^{(i)} \geq 0, \Sigma_{i=1}^{n_Q} w_Q{}^{(i)} = 1)$.*

- *A distance function $\mathcal{D}_Q$ which, given a point O in the space S, returns the distance between the query and the point. To compute the overall distance, we use a point to point distance function D_Q which, given two points in S, returns the distance between them. We assume D_Q to be a weighted L_p metric, i.e., for a given value of p, the distance between two points T_1 and T_2 in S is given by:*[4]

$$D_Q(T_1, T_2) = [\Sigma_{j=1}^{d_S} \mu_Q{}^{(j)} (|T_1[j] - T_2[j]|)^p]^{1/p} \tag{8.1}$$

where $\mu_Q{}^{(j)}$ denotes the weight associated with the jth dimension of S. $(1 \geq \mu_Q{}^{(j)} \geq 0, \Sigma_{j=1}^{d_S} \mu_Q{}^{(j)} = 1)$. D_Q specifies which L_p metric to use (i.e., the value of p) and the values of the dimension weights. We use the point to point distance function D_Q to construct the aggregate distance function $\mathcal{D}_Q(Q, O)$ between the multiple query points ($\mathcal{P}_Q$) and the object O (in S) $\mathcal{D}_Q(Q, O)$ is the aggregate of the distances between O and the individual points $P_Q{}^{(i)} \in \mathcal{P}_Q$.

$$\mathcal{D}_Q(Q, O) = \sum_{i=1}^{n_Q} w_Q{}^{(i)} D_Q(\mathcal{P}_Q^{(i)}, O) \tag{8.2}$$

We use weighted sum as the aggregation function but any other function can be used as long as it is weighted and monotonic [46]. ■

The choice of $\mathcal{D}_Q$ (i.e., the choice of the L_p metric) and the intra-feature weights captures the user perception within the space. We next describe how we choose the weights/metric.

8.2.3 Query Refinement

The intra predicate query refinement approaches described in section 7.4 apply to the multipoint queries used here. Specifically, we use the query point movement (QPM) and query expansion (QEX) approaches, both in conjunction with re-weighting of dimensions as described in section 7.4 for the remainder of this chapter. QPM converts the set $\mathcal{P}_Q$ to a centroid and changes its position to capture the query. QEX adds or removes points from the set $\mathcal{P}_Q$. Re-weighting modifies the weights in $\mathcal{W}_Q$ and applies to selection and join queries. Note that how the query points/weights

[4]Note that this assumption is general since most commonly used distance functions (e.g., Manhattan distance, Euclidean distance, Bounding Box distance) are special cases of the L_p metric. However, this excludes distance functions that involve "cross correlation" among dimensions. Handling cross-correlated functions has been addressed in [138] and can be incorporated to the techniques developed in this chapter.

```
variable $queue$ : MinPriorityQueue($object$ | $node$)
function GetNext($Q$)
1  while not $queue$.IsEmpty() do
2      top=$queue$.Pop();
3      if top is an object
4          return top;
5      else if top is a leaf node
6          for each object $O$ in top
7              $queue$.push($O$, $\mathcal{D}_Q(Q, O)$);
8      else /* top is an index node */
9          for each child node $N$ in top
10             $queue$.push($N$, $MINDIST(Q, N)$);
11     endif
12 enddo
end function
```

Table 8.2: Basic Incremental k-NN Algorithm

in $\mathcal{P}_Q$ and $\mathcal{W}_Q$ are updated is inconsequential to the discussion in the rest of the chapter. *This chapter discusses how to evaluate the refined query after it has been constructed using one of the above models: hence, the techniques presented here will work irrespective of how it was constructed.*

8.2.4 Query Evaluation

We first describe how a 'starting' query is evaluated.

8.2.4.1 Selection Queries

One option is to use sequential scan followed by sorting to return the next best object. This technique is prohibitively expensive when the database is large (see figure 8.12).

There are several approaches to compute the nearest neighbors of a spatial object or point [131, 71, 6]. Given the constraints of the application area we envision, the following properties are desirable in an algorithm:

- it works with data partitioning indexing structures that support the MINDIST operator (e.g., R-tree [65], HybridTree [25])
- the output is sorted: it returns the nearest neighbors ordered by their distance to the query
- the algorithm is easily implemented in a pipeline fashion and has no a priori restriction on the largest distance or number of neighbors to be returned
- avoids the potential of query restarts inherent in the arbitrary selection of an epsilon for a range query

An algorithm that fulfills these requirements is an incremental nearest neighbor algorithm based on the work of Roussopoulos [131] and Hjaltson [71].

Basic k-NN algorithm: The algorithm is shown in table 8.2. It maintains a priority queue that contains index nodes as well as data objects prioritized based on their distance from the query Q, i.e., the smallest item (either node or object) always appears at the top of the queue (min-priority queue). Initially, the queue contains only the root node (before the first $GetNext(Q)$ is invoked). At each step, the algorithm pops the item from the top of the queue: if it is an object, it is returned to the caller; if it is a node, the algorithm computes the distance of each of its children from the query and pushes it into the queue. The distances are computed as follows: if the child is a data object, the distance is that between the query and the object point; if the child is a node, the distance is the minimum distance (referred to as MINDIST [70]) from the query point to the nearest (according to the distance function) boundary of the node.

8.2.4.2 Multidimensional Join

A straightforward option is to use a nested loop join, calculate the distance between all objects, sort the result and iteratively return the next best answer. This option can become very expensive when the two relations involved are large since the result size is proportional to the multiplication of the size of the individual relations. This negates its use for interactive querying. Among the many available algorithms for multidimensional join, the incremental algorithm by Hjaltson and Samet [72] stands out for satisfying several desirable properties:

- it is similar in spirit to the incremental Nearest Neighbor selection algorithm we use
- it works with data partitioning indexing structures that support the MINDIST operator (e.g., R-tree [65], HybridTree [25])
- the output is sorted: it returns the closest pairs first, followed by more distant pairs
- the algorithm is easily implemented in a pipeline fashion and has no a priori restriction on the largest distance or number of pairs to be returned
- it is optimized for the case where the number of pairs shown to the user is small as compared to other algorithms that must first compute the full result, then sort it before being presented to the user

For these reasons, we used this algorithm as a basic building block to develop an algorithm for efficient re-evaluation of refined queries. The algorithm is shown in figure 8.3. Hjaltson presented several variations of the join algorithm which differ in the policy for navigating the trees [72]. Some policies give preference to a depth-first approach, while a symmetric approach is more breath-first in nature. Regardless of the traversal policy, the algorithm uses a priority queue to process a pair of nodes or data items at a time. The priority queue returns the closest pair of $\langle node,\ node\rangle$, $\langle node,\ point\rangle$, $\langle point,\ node\rangle$, $\langle point,\ point\rangle$ seen so far. If the pair consists of data items only, it is returned with the corresponding distance. Else, the pair is explored (refined) into its components for which new distances are computed and are added to the priority queue. The algorithm also

```
type { ($node$ | $point$) : A, ($node$ | $point$) : B} : pair
variable queue: MinPriorityQueue(distance, pair)
function GetNextPair()
1  while not queue.IsEmpty() do
3      top=queue.Pop();
4      if top.A and top.B are objects /*i.e., points*/
5          return top;
6      else if top.A and top.B are nodes
7          for each child $o_1$ in top.A
8              for each child $o_2$ in top.B
10                 if $o_1$ and $o_2$ are nodes
11                     queue.insert($MINDIST\_RECT(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
12                 if $o_1$ is a node and $o_2$ is a point
13                     queue.insert($MINDIST(o_2, o_1)$, $\langle o_1, o_2 \rangle$);
14                 if $o_1$ is a point and $o_2$ is a node
15                     queue.insert($MINDIST(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
16                 if $o_1$ is a node and $o_2$ is a point
17                     queue.insert($\mathcal{D}_Q(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
18     else if top.A is a node and top.B is a point
19         for each child $o_1$ in top.A
20             if $o_1$ is a node
21                 queue.insert($MINDIST(top.B, o_1)$, $\langle o_1, top.B \rangle$);
22             if $o_1$ is a point
23                 queue.insert($\mathcal{D}_Q(o_1, top.B)$, $\langle o_1, top.B \rangle$);
24     else if top.A is a point and top.B is a node
25         /* mirror of above with top.A and top.B reversed*/
26     else if top.A is a point and top.B is a point
27         queue.insert($\mathcal{D}_Q(top.A, top.B)$, $\langle top.A, top.B \rangle$);
28     endif
29 enddo
end function
```

Table 8.3: Basic Incremental Multidimensional Join Algorithm

incorporates a pruning option to limit the maximum distance between pairs which in turn influences the size of the priority queue. We describe the $MINDIST\_RECT$ later in section 8.5.

8.2.5 Problem Statement

In this chapter we address the following problems: (1) Given a table with a multidimensional column (attribute) A and a refined selection query Q over that column, how to evaluate the selection and return $answer(Q)$ as *efficiently* as possible, and (2) given two tables R and T each with a multidimensional column (attribute) $R.A$ and $T.A$, and a refined query Q, how to evaluate the join of R and T and return $answer(Q)$ as *efficiently* as possible. As mentioned before, since Q is given to us (by the refinement model), the techniques proposed in the chapter have no effect on the quality of the answers, i.e., on $answer(Q)$; the only goal is efficiency. To achieve the goal, we need to address the following problems:

- **Multipoint queries and arbitrary distance functions**: The $GetNext(Q)$ operation is performed by executing the k-NN algorithm on the corresponding F-index Idx. Traditionally, the k-NN algorithm has been used for single point queries, i.e., Q is a single point in S and the Euclidean distance function, i.e., $\mathcal{D}_Q$ is Euclidean (no dimension weights) [131, 70]. In a query refinement environment, the above assumptions do not hold. We discuss how we implement $GetNext(Q)$ efficiently when (1) Q can be a multipoint query and (2) $\mathcal{D}_Q$ can be any L_p metric and can have arbitrary dimension weights.[5] We discuss this in section 8.3.

- **Optimization of refined selection queries**: The multipoint query optimization is a necessary building block for our techniques to execute refined selection queries efficiently. We focus our work on first achieving I/O optimality and then to further improve by achieving computational optimality. We show that in general it is not possible to achieve computational optimality under the design constraints (e.g., incremental processing), and propose a heuristic approach to get close to the optimality criteria. We present our techniques in section 8.4.

- **Optimization of refined join queries**: We extend the techniques applied to refined selection queries to refined join queries. We focus on reducing the I/O and CPU requirements of joins.

[5]Note that it is possible to avoid multipoint queries (i.e., support only single point queries) by always using QPM as the query modification technique. However, Porkaew et. al. show that QEX based techniques usually perform better than QPM based ones in terms of retrieval effectiveness [123, 122]. Hence supporting multipoint queries efficiently is important for effective and efficient query refinement. Also, since multipoint queries are a generalization of single point queries, supporting multipoint queries makes the techniques developed in this chapter applicable irrespective of the query modification technique used.

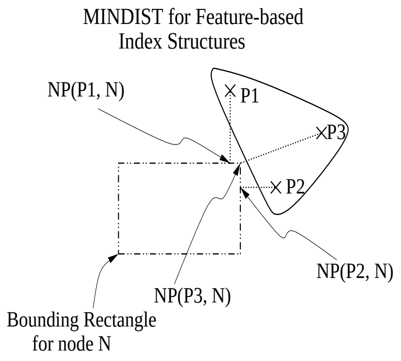

MINDIST for Feature-based Index Structures
NP(P1, N)
P1
P3
P2
NP(P2, N)
NP(P3, N)
Bounding Rectangle for node N

Figure 8.1: MINDIST Computation

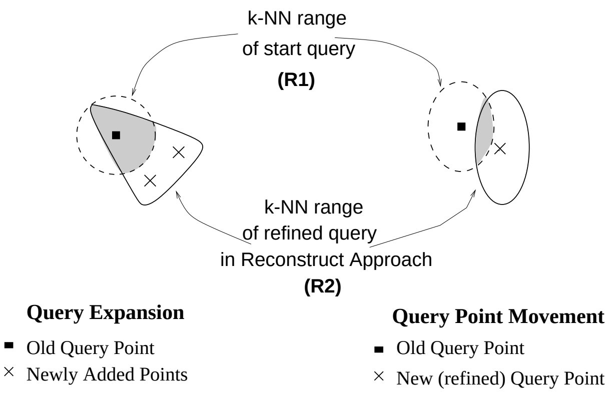

k-NN range of start query
(R1)
k-NN range of refined query in Reconstruct Approach
(R2)
Query Expansion
Old Query Point
Newly Added Points
Query Point Movement
Old Query Point
New (refined) Query Point

Figure 8.2: I/O Cost of Naive and Reconstruction Approaches

8.3 k-Nearest Neighbor Selection Algorithm for Multipoint Queries

The $GetNext(Q)$ function in table 8.2 can handle only single point queries, i.e., $n_Q = 1$. One way to handle a multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q \rangle$ is by incrementally determining the next nearest neighbor $NN_Q^{(i)}$ of each point $P_Q^{(i)} \in \mathcal{P}_Q$ by invoking $GetNext(P_Q^{(i)})$ (for $i = [1, n_Q]$) computing its overall distances $\mathcal{D}_Q(Q, NN_Q^{(i)})$ (using Equation 8.2) and storing them in a buffer until we are sure that we have found the next nearest neighbor to Q. This technique, referred to as the *Multiple Expansion Approach*, was proposed in our early work [125] and by FALCON [169] which showed it to perform better than other approaches like the centroid-based expansion and single point expansion approaches [125].

In this chapter, we propose an alternate approach to evaluating multipoint queries. We refer to it as the *multipoint approach*. In this approach, we modify the $GetNext(Q)$ function to be able to handle multipoint queries, i.e., it should be able to compute the distances of the nodes and objects directly from the multipoint query Q and explore the priority queue based on those distances. The basic algorithm in table 8.2 does not change, the only changes are the distance computations. As before, there are 2 types of distance computations: (1) distance of an object to the multipoint query, i.e., $\mathcal{D}_Q(Q, O)$ (2) distance of a node to the multipoint query (MINDIST), i.e., $MINDIST(Q, N)$. The computation of (1) follows directly from Equation 8.2. So all we need to do is to define the distance $MINDIST(Q, N)$ between a multipoint query Q and a node N of the F-index. As in the single point case, the definition of MINDIST depends on the shape of the node boundary. We assume that the F-index Idx is a feature-based index (e.g. Hybrid tree, R-tree, X-tree, KDB-tree, hB-tree) because distance-based index structures (e.g., SS-tree, SR-tree, TV-tree, M-tree) cannot handle arbitrary dimension weights. In a feature-based index, the bounding region (BR) of each node N is always, either explicitly or implicitly, a d_S-dimensional rectangle in the feature space S [25].

Definition 8.3.1 (MINDIST for Multipoint Queries) *Given the bounding rectangle (BR)* $R_N =$

$\langle L, H \rangle$ of a node N, where $L = \langle l_1, l_2, ..., l_{d_S} \rangle$ and $H = \langle h_1, h_2, ..., h_{d_S} \rangle$ are the two endpoints of the major diagonal of R_N, $l_i \le h_i$ for $1 \le i \le d_S$. The nearest point $NP(P_Q^{(i)}, N)$ in R_N to each point $P_Q^{(i)}$ in the multipoint query $Q = \langle n_Q, \mathcal{P}_Q, \mathcal{W}_Q, \mathcal{D}_Q \rangle$ is defined as follows (explained in figure 8.1).

$$NP(P_Q^{(i)}, N)[j] = \begin{cases} l_j & \text{if } P_Q^{(i)}[j] < l_j \\ h_j & \text{if } P_Q^{(i)}[j] > h_j \\ P_Q^{(i)}[j] & \text{otherwise} \end{cases} \tag{8.3}$$

where $NP[j]$ denotes the position of NP along the jth dimension of the feature space S, $1 \le j \le d_S$. $MINDIST(M, N)$ is defined as:

$$MINDIST(Q, N) = \sum_{i=1}^{n_Q} w_Q^{(i)} \mathcal{D}_Q(P_Q^{(i)}, NP(P_Q^{(i)}, N)) \tag{8.4}$$

■

The *GetNext* function can handle arbitrary distance functions $\mathcal{D}_Q$ (i.e., L_p metrics with arbitrary intra-feature weights). The above algorithm is correct (i.e., $GetNext(Q)$ returns the next nearest neighbor of Q) if $MINDIST(Q, N)$ always *lower bounds* $\mathcal{D}_Q(Q, T)$ where T is any point stored under N.

Lemma 8.3.1 (Correctness of GetNext algorithm) *$MINDIST(Q, N)$ lower bounds $\mathcal{D}_Q(Q, T)$.*

Proof: Let N be a node of the index structure, R be the corresponding bounding rectangle and T be any object under N. Let us assume that $\mathcal{D}_Q$ is monotonic. We need to show that $MINDIST(Q, N) \le \mathcal{D}_Q(Q, T)$. Let $T = \langle t_1, t_2, ..., t_{d_S} \rangle$ be the be the d_S-dimensional vector and $R = \langle l_1, l_2, ..., l_{d_S}, h_1, h_2, ..., h_{d_S} \rangle$ be the d_S-dimensional bounding rectangle. Since T is under N, T must be spatially contained in R.

$$l_j \le t_j \le h_j, j = [1, d_S] \tag{8.5}$$

Using the definition of $NP(P_Q^{(i)}, N)[j]$ and $\mathcal{D}_Q(P_Q^{(i)}, T)$, Equation 8.5 implies

$$|P_Q^{(i)}[j] - NP(P_Q^{(i)}, N))[j]| \le |P_Q^{(i)}[j] - T[j]|, j = [1, d_S] \tag{8.6}$$

$$\Rightarrow \mathcal{D}_Q(P_Q^{(i)}, NP(P_Q^{(i)}, N)) \le \mathcal{D}_Q(P_Q^{(i)}, T) \quad \text{since } \mathcal{D} \text{ is a weighted } L_p \text{ metric} \tag{8.7}$$

$$\Rightarrow \sum_{i=1}^{n} w_i \mathcal{D}_Q(P_Q^{(i)}, NP(P_Q^{(i)}, N)) \le \sum_{i=1}^{n} w_Q^{(i)} \mathcal{D}_Q(P_Q^{(i)}, T) \quad \text{since } w_i \ge 0 \tag{8.8}$$

$$\Rightarrow MINDIST(Q, N) \le D_Q(Q, T) \tag{8.9}$$

■

Our experiments show that the multipoint approach is significantly more efficient compared to the previously proposed multiple expansion approach (cf. section 8.6).

8.4 Evaluation of k-NN Refined Selection Queries

A naive way to evaluate a single feature refined query is to treat it just like a starting query and execute it from scratch as discussed in section 8.2.4. This approach is wasteful as we can save most of the execution cost of the refined query, both in terms of disk accesses (I/O cost) and distance calculations (CPU cost), by exploiting information generated during the previous iterations of the query. In this section, we discuss how to optimize the $GetNext(Q)$ function. In the naive approach, the same nodes of Idx may be accessed from scratch by the k-NN algorithm repeatedly iteration after iteration. In other words, a node of Idx is being accessed from disk multiple times during the execution of a query (over several iterations) causing unnecessary disk I/O. For example, let us consider the query shown in figure 8.2. Region $R1$ represents the iso-distance range corresponding to the distance of the kth NN of the starting query – it is the region in the feature space already explored to return the top k matches to the starting query, i.e., all nodes overlapping with $R1$ were accessed and all objects (k in number) in that region were returned. $R2$ represents the iso-distance range corresponding to the distance of the kth NN of the refined query – it is the region that needs to be explored to answer the refined query, i.e., all nodes overlapping with $R2$ need to be explored and all objects (k in number) in that region need to be returned. If no buffering is used, to evaluate the refined query, the naive approach would access *all* the nodes overlapping with $R2$ from the disk, thus accessing those nodes that overlap with the shaded region from the disk twice. If traditional LRU buffering is used, some of the nodes overlapping with the shaded region may still be in the database buffer and would not require disk accesses during the evaluation of the refined query. Our experiments show that, for reasonable buffer sizes, most of the nodes in the shaded region get ejected from the buffer before they are accessed by the refined query. This is due to the use-recency based page replacement policy used in database buffers. The result is that the naive buffering approach still needs to perform large numbers of repeated disk accesses to evaluate refined queries (cf. figure 8.9). Concurrent query sessions by different users and user think time between iterations would further reduce buffer hit ratio, causing the naive approach to perform even more poorly.

Our goal in this chapter is to develop a more I/O-efficient technique to evaluate refined queries. If a node is accessed from disk once during an iteration of a query, we should keep it in main memory (by caching) so that it is not accessed from disk again in any subsequent iteration of that query. In figure 8.2, to evaluate the refined query, an I/O optimal algorithm would access from disk only those nodes that overlap with region $R2$ but do not overlap with region $R1$ and access the remaining nodes (i.e., those that overlap with the shaded region) from memory (i.e., from the cache). We formally state I/O optimality as follows.

Definition 8.4.1 (I/O Optimality) *Let N be a node of the F-index Idx. An algorithm executing the refined query Q^{new} is I/O optimal if it makes a disk access for N* only if *(1) there exists at least one desired answer O such that $\mathcal{D}_Q(Q^{new}, O) \geq MINDIST(Q^{new}, N)$ and (2) N is not accessed during any previous iteration (say Q^{old}).* ■

Condition (1) is necessary to guarantee no false dismissals, while Condition (2) guarantees that a node is accessed from the disk at most once throughout the execution of the query across all iterations of refinement. The naive approach is not I/O optimal as it does not satisfy condition (2).

We achieve I/O optimality by caching on a per-query basis instead of using a common LRU buffer for all queries. To ensure condition (2), we need to 'cache' the contents of each node accessed by the query in any iteration and retain them in memory till the query ends (i.e., till the last iteration) at which time the cache can be freed and the memory can be returned to the system. Since the priority queue generated during the execution of the starting query contains the information about the nodes accessed, we can achieve the above goal by caching the priority queue. We assume that, for each item (node or object), the priority queue stores the feature values of the item (i.e., the bounding rectangle if the item is a node, the feature vector if it is an object) in addition to its id and its distance from the query. This is necessary since, in some approaches, we need to recompute the distances of these items based on the refined queries. We also assume that the entire priority queue fits in memory as is commonly assumed in other works on nearest neighbor retrieval [86, 139].[6] In the following subsections, we describe how we can use the priority queue generated during the starting query (that contains items ordered based on distance from the starting query) to efficiently obtain the top k matches based on their distances from the refined query.

8.4.1 Full Reconstruction (FR)

We first describe a simple approach called the full reconstruction approach. In this approach, to evaluate the refined query, we 'reconstruct' a new priority queue $queue_{new}$ from the cached priority queue $queue_{old}$ by popping each item from $queue_{old}$, recomputing its distance from the refined query Q^{new}, and then pushing it into $queue_{new}$. We discard the old queue when all items have been transferred. We refer to this phase as the *transfer phase*. Subsequently, the multipoint k-NN algorithm proposed in section 8.3 is invoked with Q^{new} as the query on $queue_{new}$. We refer to this phase as the *explore phase*. The queue is handed from iteration to iteration through the reconstruction process, i.e., the $queue_{new}$ of the previous iteration becomes the $queue_{old}$ of the current iteration and is used to construct the $queue_{new}$ of the current iteration. Thus, if a node is accessed once, it remains in the priority queue for the rest of the query and in any subsequent iteration, is accessed directly from the queue (which is assumed to be entirely in memory) instead

[6]This assumption is reasonable when the number k of top matches requested is relatively small compared to the size of the database which is usually the case [21]. For example, in our experiments, when $k = 100$, the size of priority queue varies between 512KB and 640KB (the size of the F-index is about 11MB). The techniques proposed in the chapter would work even if the priority queue does not fit in memory and would still perform better than the naive approach; however, they may not be I/O optimal.

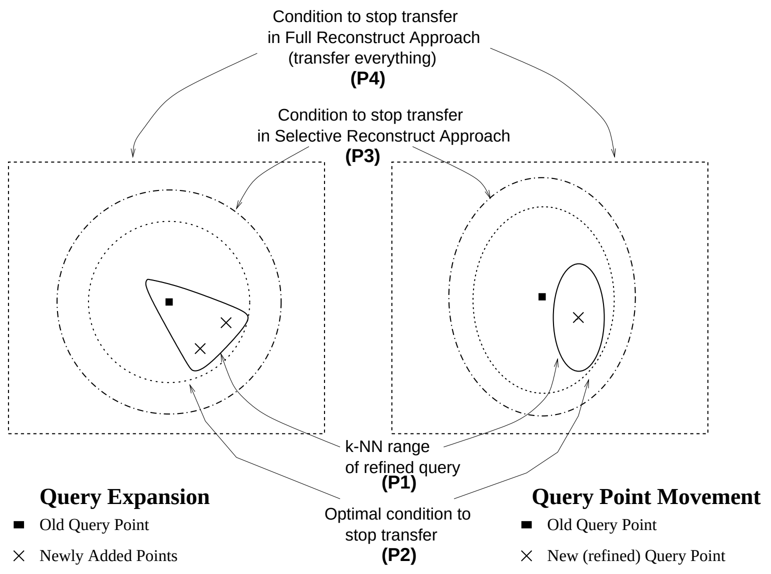


Figure 8.3: CPU Cost of Full and Selective Reconstruction Approaches

of reloading from the disk. The entire sequence of iterations is managed using two queues and swapping their roles (old and new) from iteration to iteration. It is easy to see that this approach is I/O optimal.

Now that we have achieved I/O optimality, let us consider the CPU cost of the approach. The CPU cost is proportional to the number of distance computations performed. The algorithm performs distance computations during the transfer phase (one computation each time an item is transferred from $queue_{old}$ to $queue_{new}$) and also during the explore phase (one computation for each child in the node being explored). Since the technique is I/O optimal, there is no node being explored unnecessarily, i.e., we cannot save any distance computations in the explore phase (already optimal). However, we can reduce the CPU cost of the transfer phase if we avoid transferring each and every item from $queue_{old}$ to $queue_{new}$ without sacrificing correctness. In other words, we should transfer only those items that are *necessary* to transfer to avoid false dismissals.

8.4.2 Selective Reconstruction (SR)

Our Selective Reconstruction approach transfers incrementally only as many items as are needed to produce the *next* result. We first describe how to achieve this within a single feedback iteration and describe the problems found, then extend it to function properly for multiple iterations and discuss the limits of this technique.

8.4.2.1 Single Iteration Selective Reconstruction

We modify the FR approach to reduce the CPU cost while retaining the I/O optimality. We only transfer those items from $queue_{old}$ to $queue_{new}$ that are necessary to ensure correctness. We achieve this by imposing a *stopping condition* on the transfer as illustrated in figure 8.3. Suppose that $GetNext$ is returning the kth NN of Q^{new}. $P1$ is the iso-distance range corresponding to the distance of the kth NN of Q^{new}, i.e., any object outside this region is at least as far as the kth NN. $P2$ is the smallest iso-distance range from Q^{old} totally containing $P1$. Therefore, to return the kth NN of Q^{new}, we can stop transferring when we reach the range $P2$ in $queue_{old}$ without sacrificing correctness since at this point, any object left unexplored in $queue_{old}$ is at least as far from Q^{new} as the kth NN of Q^{new}. This represents the *optimal* stopping condition because we cannot stop before this without sacrificing correctness. In other words, if we stop before this, there will be at least one false dismissal. The FR approach has no stopping condition at all as is graphically shown in figure 8.3 by region $P4$, i.e., the entire feature space.

While in the FR approach, the entire transfer phase is followed by the entire explore phase, here the two phases are interspersed. The algorithm pseudo-code is shown in table 8.4. During the transfer phase, we transfer items from $queue_{old}$ to $queue_{new}$ until we are sure that no item exists in $queue_{old}$ that is closer to Q^{new} than the top item of $queue_{new}$ (i.e., the next item to be explored – top_{new}). In other words, the lower bounding distance $LBD(queue_{old}, Q^{new})$ of any item in $queue_{old}$ is greater than or equal to the distance between top_{new} and Q^{new} ($top_{new}.distance = \mathcal{D}_Q(Q^{new}, top^{new})$), this is the stopping condition in Line 5. If we are sure, we go into the explore phase, i.e., we explore top_{new}. If top_{new} is an object, it is *guaranteed* to be the next best match to Q^{new} and is returned to the caller. Otherwise, it is a node; in that case, we access it from disk, compute the distance of each child from Q^{new} and push it back into $queue_{new}$; then we return to the transfer phase. It is easy to see that the selective reconstruction technique is I/O optimal.

Let us now explain the notion of LBD in the stopping condition (in Line 5). $LBD(queue_{old}, Q^{new})$ denotes the the smallest distance any item in $queue_{old}$ can have with respect to the new query Q^{new}, i.e., $LBD(queue_{old}, Q^{new}) \leq \mathcal{D}_Q^{new}(Q^{new}, O)$ for all $O \in queue_{old}$. To complete the algorithm, we need to specify how to compute $LBD\ (queue_{old}, Q^{new})$. We need to compute it without incurring *any* extra cost, i.e., in a single constant time operation. If we can obtain the optimal $LBD(queue_{old}, Q^{new})$ (i.e., there exists an object $O \in queue_{old}$ where $LBD(queue_{old}, Q^{new}) = \mathcal{D}_Q^{new}(Q^{new}, O)$), we can achieve the optimal stopping condition (shown by $P2$ in figure 8.3) and hence the optimal CPU cost.

Example 8.4.1 (SR Algorithm) *Figure 8.4 shows old and new queries (Q^{old} and Q^{new}) with iso-distance curves at one unit intervals. For example, in the figure, the outermost iso-distance curve of Q^{new} is the optimal stopping condition for any points within the first iso-distance curve of Q^{old} (i.e., points A, B, C).*
Following figure 8.4, the SR algorithm examines $queue_{old}$ in distance order. It first compares the top item from $queue_{new}$ which at first is empty (and thus 0) to the LBD of point A, and finds that the condition is not met, A is then transferred to $queue_{new}$. This continues until E is retrieved from $queue_{old}$. For E, the optimal stopping condition should be true as there are no items in $queue_{old}$ that can possibly be closer to Q_{new}. Thus E would be returned by GetNext.

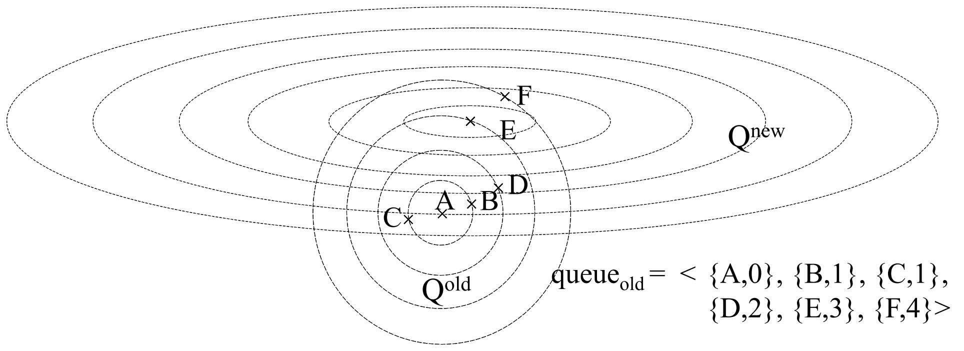


Figure 8.4: LBD example

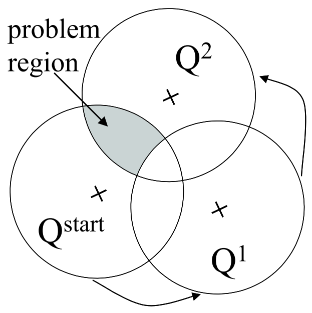


Figure 8.5: Problem Region for Single-iteration SR Approach

```
variable $Q^{old}$: Query // (of previous iteration)
variable $queue_{old}$: MinPriorityQueue // (of previous iteration)
variable $queue_{new}$: MinPriorityQueue // (of current iteration)
function GetNext($Q^{new}$)

1   while (not $queue_{new}$.IsEmpty()) do
          /* TRANSFER PHASE (Lines 2-10)*/
2         while (not $queue_{old}$.IsEmpty()) do
3               $top_{old}$=$queue_{old}$.Top();
4               $top_{new}$=$queue_{new}$.Top();
                /* STOPPING CONDITION (Line 5)*/
5               if ($LBD(queue_{old}, Q^{new}) \geq top_{new}.distance$) break
6               else $queue_{old}$.Pop();
7                     Recompute $top_{old}$.distance based on $Q^{new}$
8                     Push $top_{old}$ into $queue_{new}$
9               endif
10        enddo
          /* EXPLORE PHASE (Lines 11-21) */
11        $queue_{new}$.Pop();
12        if $top_{new}$ is an object
13           return $top_{new}$;
14        else if $top_{new}$ is a leaf node
15           for each object in $top_{new}$
16                $queue_{new}$.push(object, $\mathcal{D}_Q(Q^{new}$, object));
17        else /* $top_{new}$ is an index node */
18           for each child of $top_{new}$
19                $queue_{new}$.push(child, $MINDIST(Q^{new}$, child));
20        endif
21  enddo
end function
```

Table 8.4: The Single-iteration GetNext Algorithm for Refined Queries using the SR Approach

Lemma 8.4.1 (CPU Optimality) *The number of distance computations (i.e., the stopping condition) is optimal if $LBD(queue_{old}, Q^{new})$ is optimal.*

Proof: Let I_{new} be an item in $queue_{new}$. Let I_{old} be the item at the top of $queue_{old}$. Assuming $LBD(queue_{old}, Q^{new})$ is tight, we need to show that if I_{old} is transferred to $queue_{new}$, there exists an item I'_{old} in $queue_{old}$ such that $\mathcal{D}_Q^{old}(I_{old}, Q^{old}) \leq \mathcal{D}_Q^{old}(I'_{old}, Q^{old})$ and $\mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \leq \mathcal{D}_Q^{new}(I_{new}, Q^{new})$. Let us assume I_{old} is transferred, i.e, the stopping condition is not satisfied ($LBD(queue_{old}, Q^{new}) < top_{new}.distance$). Since LBD is tight, there exists an unexplored item I'_{old} in $queue_{old}$ such that $\mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \leq top_{new}.distance$. Since $top_{new}.distance \leq \mathcal{D}_Q^{new}(I_{new}, Q^{new})$, $\mathcal{D}_Q^{new}(I'_{old}, Q^{new}) \leq \mathcal{D}_Q^{new}(I_{new}, Q^{new})$. Since I_{old} is at the top of $queue_{old}$ and I'_{old} is yet unexplored, $\mathcal{D}_Q^{old}(I_{old}, Q^{old}) \leq \mathcal{D}_Q^{old}(I'_{old}, Q^{old})$. ■

We believe that it is not possible to compute the optimal LBD without exploring $queue_{old}$ (which would defeat the purpose of the reconstruction approach). Instead we derive a conservative estimate of LBD. For correctness, the estimated LBD must be an LBD, i.e., it must always under-estimate the optimal LBD. The closer the estimate to the optimal LBD, the fewer the number of transfers, and the lower the CPU cost. In the derivation, we exploit the fact that $top_{old}.distance$ lower bounds the distance of any item in $queue_{old}$ from the *old* query Q^{old}. For simplicity of notation, we define the distance $D(Q^{old}, Q^{new})$ between two multipoint queries as follows. Let $Q^{old} = \langle n_Q^{old}, \mathcal{P}_Q^{old}, \mathcal{W}_Q^{old}, \mathcal{D}_Q^{old} \rangle$ be the old multipoint query where $\mathcal{P}_Q^{old} = \{P_Q^{old(1)}, \cdots, P_Q^{old(n_Q^{old})}\}$ and $\mathcal{W}_Q^{old} = \{w_Q^{old(1)}, \cdots, w_Q^{old(n_Q^{old})}\}$. Let $Q^{new} = \langle n_Q^{new}, \mathcal{P}_Q^{new}, \mathcal{W}_Q^{new}, \mathcal{D}_Q^{new} \rangle$ be the new multipoint query where $\mathcal{P}_Q^{new} = \{P_Q^{new(1)}, \cdots, P_Q^{new(n_Q^{new})}\}$ and $\mathcal{W}_Q^{new} = \{w_Q^{new(1)}, \cdots, w_Q^{new(n_Q^{new})}\}$. The distance $D(Q^{old}, Q^{new})$ is defined as the weighted all-pairs distance between the constituent points:

$$D(Q^{old}, Q^{new}) = \sum_{i=1}^{n_Q^{old}} \left(w_Q^{old(i)} \sum_{j=1}^{n_Q^{new}} w_Q^{new(j)} \mathcal{D}_Q^{new}(P_Q^{old(i)}, P_Q^{new(j)}) \right) \tag{8.10}$$

Also, let $\mu_Q^{old(k)}, k = [1, d_S]$ and $\mu_Q^{new(k)}, k = [1, d_S]$ denote the old and new intra-feature weights respectively:

$$\mathcal{D}_Q^{old}(T_1, T_2) = [\Sigma_{j=1}^{d_S} \mu_Q^{old(j)} (|T_1[j] - T_2[j]|)^p]^{1/p} \tag{8.11}$$

$$\mathcal{D}_Q^{new}(T_1, T_2) = [\Sigma_{j=1}^{d_S} \mu_Q^{new(j)} (|T_1[j] - T_2[j]|)^p]^{1/p} \tag{8.12}$$

The following lemma defines the stopping condition (SC) in Line 5.

Lemma 8.4.2 (Stopping Condition) *$(\frac{top_{old}.distance}{K} - D(Q^{new}, Q^{old}))$ lower bounds the distance of any unexplored object in $queue_{old}$ from Q^{new} where $K = max_{k=1}^{d_S} \left(\frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}} \right)$.*

Proof: Let I be any item in $queue_{old}$. We need to show $\mathcal{D}_Q^{old}(I, Q^{old}) \geq top_{old}.distance \Rightarrow \mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{top_{old}.distance}{max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old})$. Let us assume

$$\mathcal{D}_Q^{old}(Q^{old}, I) \geq top_{old}.distance \tag{8.13}$$

Let $P_Q^{old(i)} \in P_Q^{old}$ and $P_Q^{new(j)} \in P_Q^{new}$. By triangle inequality,

$$\mathcal{D}_Q^{new}(P_Q^{new(j)}, I) \geq \mathcal{D}_Q^{new}(P_Q^{old(i)}, I) - \mathcal{D}_Q^{new}(P_Q^{new(j)}, P_Q^{old(i)}) \tag{8.14}$$

From Equations 8.11 (assuming $\mathcal{D}$ is monotonic),

$$\mathcal{D}_Q^{new}(P_Q^{old(i)}, I) \geq \frac{\mathcal{D}_Q^{old}(P_Q^{old(i)}, I)}{max_{k=1}^{d_S} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}} \tag{8.15}$$

From Equation 8.14 and 8.15,

$$\mathcal{D}_Q^{new}(P_Q^{new(j)}, I) \geq \frac{\mathcal{D}_Q^{old}(P_Q^{old(i)}, I)}{K} - \mathcal{D}_Q^{new}(P_Q^{new(j)}, P_Q^{old(i)}) \tag{8.16}$$

where $K = max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}$. Multiplying both sides by $w_Q^{old(i)}$ (since $w_Q^{old(i)} \geq 0$) and summing over $i = 1$ to n_{old},

$$\sum_{i=1}^{n_{old}}(w_Q^{old(i)}\mathcal{D}_Q^{new}(P_Q^{new(j)}, I)) \geq \frac{1}{K}\sum_{i=1}^{n_{old}}(w_Q^{old(i)}\mathcal{D}_Q^{old}(P_Q^{old(i)}, I)) - \sum_{i=1}^{n_{old}}(w_Q^{old(i)}\mathcal{D}_Q^{new}(P_Q^{new(j)}, P_Q^{old(i)})) \tag{8.17}$$

$$\Rightarrow \mathcal{D}_Q^{new}(P_Q^{new(j)}, I) \geq \frac{\mathcal{D}_Q^{old}(Q^{old}, I)}{K} - \mathcal{D}_Q^{new}(P_Q^{new(j)}, Q^{old}) \tag{8.18}$$

Multiplying both sides by $w_Q^{new(j)}$ (since $w_Q^{new(j)} \geq 0$) and summing over $j = 1$ to n_{new},

$$\sum_{j=1}^{n_{new}}(w_Q^{new(j)}\mathcal{D}_Q^{new}(P_Q^{new(j)}, I)) \geq \frac{1}{K}\sum_{j=1}^{n_{new}}(w_Q^{new(j)}\mathcal{D}_Q^{old}(Q^{old}, I)) - \sum_{j=1}^{n_{new}}(w_Q^{new(j)}\mathcal{D}_Q^{new}(P_Q^{new(j)}, Q_Q^{old})) \tag{8.19}$$

$$\Rightarrow \mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{\mathcal{D}_Q^{old}(Q^{old}, I)}{K} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old}) \tag{8.20}$$

Equations 8.13 and 8.20 imply

$$\mathcal{D}_Q^{new}(Q^{new}, I) \geq \frac{top_{old}.distance}{max_{k=1}^{m} \frac{\mu_Q^{old(k)}}{\mu_Q^{new(k)}}} - \mathcal{D}_Q^{new}(Q^{new}, Q^{old}) \tag{8.21}$$

■

The $LBD(queue_{old}, Q^{new})$ in the stopping condition in Line 5 of the SR algorithm (cf. table 8.4) can now be replaced by the above estimate. Thus, the stopping condition is:

$$(\frac{top_{old}.distance}{K} - D(Q^{new}, Q^{old})) \geq top_{new}.distance \tag{8.22}$$

Note that the above SC is general. It is valid irrespective of the query modification technique used, i.e., it is valid for both QPM and QEX models. It is also valid irrespective of whether intra-feature reweighting is used or not. If intra-feature reweighting is not used (i.e., all weights are considered equal), K turns out to be 1 which means there is no effect of intra-feature reweighting at all. Also note that both K and $D(Q^{new}, Q^{old})$ are computed just once during the execution of the refined query. The computation of K is proportional only to the number of dimensions d_S in the space. The computation of LBD involves just two arithmetic operations, a division followed by a subtraction. Since we use an estimate of LBD and not the exact LBD, in general, SC is not optimal. This implies that, as shown in figure 8.3, the stopping condition is range $P3$ and not the optimal range $P2$. We perform experiments to compare the above stopping condition with the optimal one in terms of CPU cost (cf. section 8.6).

Example 8.4.2 *Consider again figure 8.4. The first item from* $queue_{old}$ *is A. Its distance to* Q^{new} *is 5, and* Q^{new} *has a high weight (0.66) in the vertical dimension and a small weight (0.33) in the horizontal dimension. So,* $K = \max\left\{\frac{0.5}{0.33}, \frac{0.5}{0.66}\right\} = \frac{0.5}{0.33} = 1.5$*, and notice that* $D(Q^{new}, Q^{old}) = 5$*. So the LBD estimate for A is* $\frac{0}{1.5} - 5 = -5$*. Likewise for point E, the LBD is* $\frac{3}{1.5} - 5 = 2 - 5 = -3$*. Here we see the lower bounding nature of our approximation. If our LBD estimate were optimal, the LBD for point E should be 0 and E could be returned.*

8.4.2.2 Multiple Iteration Selective Reconstruction

The above discussion of the SR approach considered how to evaluate the first iteration of the query refinement given the priority queue $queue_{start}$ generated during the execution of the start query (iteration 0). The discussion omitted what to do when the iteration finishes and a new feedback iteration starts. If we discard the old queue, create a new, empty queue and follow the same algorithm, we will sacrifice the correctness of the algorithm. Consider figure 8.5 which shows the starting query Q^{start} and two feedback iterations Q^1 and Q^2 using the simple QPM model and no reweighting. If we follow the SR algorithm for Q^2, we will miss answers in the *problem region.* These items were present in $queue_{start}$, but not transferred to $queue_1$ (some may be included since our LBD is an approximation), therefore they are irretrievably lost for Q^2.

To properly handle multiple feedback iterations using the SR algorithm, we must accommodate for unprocessed items in earlier iterations. Copying the remaining items to the new queue would be equivalent to the FR approach. Instead, our solution is to maintain a history of the queries and queues from the start. This approach stores the same number of items as the FR approach (no item is ever discarded), and has the benefits of the SR approach in exchange for some administrative complexity. Table 8.5 shows the algorithm that accounts for multiple iterations. We maintain a history list of tuples $\langle Q^{(i)}, queue_i, K_i, D(Q^{(i)}, Q^{new})\rangle$ that keep each query, queue and additional parameters for each iteration. When a new iteration starts, we update all tuples in the history with the corresponding new values for K_i and $D(Q^{(i)}, Q^{new})$. The algorithm then extends the single iteration SR by selecting in each iteration the minimum LBD among all queries in the history to maintain the LBD correctness. The CPU complexity of the algorithm is only increased by selecting the minimum LBD among all previous queues; as discussed above, this means two arithmetic operations per iteration in addition to the single iteration SR algorithm. The storage requirements for each iteration consist of the query Q_i which is dependent only on the number of points and the dimensionality (but typically only a few hundred bytes in size), and the values K_i and $D(Q^{(i)}, Q^{new})$ which are just two constants. The total number of items in all queues is at most the that of the FR approach (only necessary nodes are expanded, and no items are dropped).

```
variable $history$: $\langle Q^{(i)}, queue_i, K_I, D(Q^{(i)}, Q^{new}\rangle$: // query and queue history
variable $queue_{new}$: MinPriorityQueue // (of current iteration)
function GetNext($Q^{new}$)

1   while (not $queue_{new}$.IsEmpty()) do
        /* TRANSFER PHASE (Lines 2-11)*/
2       while ($\exists i \mid \neg history.queue_i$.IsEmpty()) do
3           $i$=$i$ | min{$LBD(history.queue_i, Q^{new})$};
4           $top_i$=$history.queue_i$.Top();
5           $top_{new}$=$queue_{new}$.Top();
            /* STOPPING CONDITION (Line 6)*/
6           if ($LBD(history.queue_i, Q^{new}) \geq top_{new}.distance$) break
7           else $history.queue_i$.Pop();
8               Recompute $top_i$.distance based on $Q^{new}$
9               Push $top_i$ into $queue_{new}$
10          endif
11      enddo
        /* EXPLORE PHASE (Lines 12-21) */
12      $queue_{new}$.Pop();
13      if $top_{new}$ is an object
14        return $top_{new}$;
15      else if $top_{new}$ is a leaf node
16        for each object in $top_{new}$
17            $queue_{new}$.push(object, $\mathcal{D}_Q(Q^{new}$, object));
18      else /* $top_{new}$ is an index node */
19        for each child of $top_{new}$
20            $queue_{new}$.push(child, $MINDIST(Q^{new}$, child));
21      endif
22  enddo
end function
```

Table 8.5: The Multi-iteration GetNext Algorithm for Refined Queries using the SR Approach

8.4.2.3 Cost–benefit based use of SR

As later iteration queries drift farther apart from the starting query, the storage and computation costs increase as more items and auxiliary information accumulate. We turn our attention to the problem of deciding whether a new query iteration should be evaluated by continuing the use of SR, or naively (followed by iterations using SR). We distinguish three distinct considerations involved:

- **IO cost:** From an IO perspective, the FR and SR techniques are IO optimal, therefore they always reduce the number of IOs over a naive execution regardless of the convergence properties of the feedback. When queries converge over iterations, as is generally the case [169, 125, 78], the IO count per iteration asymptotically approaches 0 (see experiments).

- **CPU cost:** With more items from early iterations lingering in the queues, the number of computations increases. An absolute upper bound on the number of iterations can be obtained by observing that each new SR iteration necessarily requires us to revise the K_i

and $D(Q^{(i)}, Q^{new})$ values for each iteration.[7] The computation of these values depends on the dimensionality of the space and the number of points in the queries. For simplicity, we estimate the cost to be the same as one distance computation. By maintaining simple statistics, we can estimate the number of distance computations used in a naive reevaluation to be the number of items for which a computation was made in the starting query Q^0. This is the number of items in $history.queue_0$ plus the k items returned to the user plus any intermediate items that were discarded $m_0^{discarded}$. A definite CPU based upper bound is then when: $k + m_0^{discarded} + \mid\ history.queue_0 \mid < i$ that is, there are more iterations than elements in the starting query.

This bound is generally much too high. A new iteration starts by exploring the current and earlier queues and transfers items until the stopping condition permits the return of a result. In practice, this generally implies several hundred distance computations that must be done after the new iteration started. SR adds computation overhead proportional to the number of iterations over FR. A better estimate can thus be obtained by deducting those initial computations from the number of iterations. Let $t^{(i)}$ be the number of items transferred during iteration i to the current queue from all earlier queues before the first result is returned to the user. Then a better estimate is: $k + m_0^{discarded} + \mid history.queue_0 \mid + t^{(i-1)} < i$. From our experiments, SR still pays off even after 50 iterations.

- **Memory cost:** To reduce the memory used, we must eliminate items from the queues. We follow a straightforward approach which drops all the queues whenever the DBMS faces a memory shortage and do a naive reexecution of the query, thus trading IO cost for a reduced memory footprint. A general algorithm to consolidate individual items under their parent node is too expensive since it must examine large portions of the queues.

8.5 Evaluation of Top-k Refined Join Queries

In this section we discuss how to optimize join queries after feedback has been submitted and processed. Due to the high cost of executing joins, it is inefficient to naively re-execute them from scratch after an iteration of relevance feedback. A naive re-execution will cause substantial unnecessary disk accesses, even when a traditional LRU database buffer cache is used due to the poor locality of the distance join algorithm.

As described in section 8.2.4.2, we use a distance based join algorithm with two index trees and a priority queue *queue* to maintain enough state to compute the next result pair. The algorithm makes use of the point to point distance function $\mathcal{D}_Q$ and the $MINDIST$ used for selection queries to compare points with tree nodes. To compare internal tree nodes to each other, we need to extend the definition of $MINDIST$ to handle two rectangles:

[7]This must be done either explicitly as in our description or implicitly if we avoid caching these values in the history.

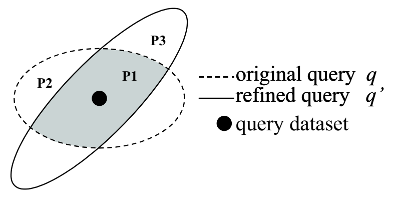


Figure 8.6: I/O overlap of original and new query

Definition 8.5.1 (MINDIST_RECT(N, N)) *Given two d dimensional bounding rectangles* $R1_N = \langle L1, H1 \rangle$ *and* $R2_N = \langle L2, H2 \rangle$ *of the nodes* N_1 *and* N_2*, where* $L = \langle l_1, l_2, ..., l_d \rangle$ *and* $H = \langle h_1, h_2, ..., h_d \rangle$ *are the two endpoints of the major diagonal of* R_N*,* $l_i \leq h_i$ *for* $1 \leq i \leq d$*.*

The nearest distance nd between the rectangles $R1$ *and* $R2$ *is defined as follows:*

$$nd[j] = \begin{cases} R1.L[j] - R2.H[j] & \text{if } R1.L[j] > R2.H[j] \\ R2.L[j] - R1.H[j] & \text{if } R1.H[j] < R2.L[j] \\ 0 & \text{otherwise} \end{cases}$$

where $nd[j]$ *denotes the nearest distance between* $R1$ *and* $R2$ *along the* j*th dimension.*

$MINDIST\_RECT(N1, N2)$ *if defined as:*

$$MINDIST\_RECT(N1, N2) = \sqrt[p]{\sum_{i=1}^{n} w_i \times |nd[i]|^p}$$

The join algorithm can handle arbitrary L_p *distance metrics with arbitrary weights for each dimension. The algorithm is correct if* $MINDIST(N1, N2)$ *always lower bounds* $DIST(T1, T2)$ *where* $T1$ *and* $T2$ *are any points stored under* $N1$ *and* $N2$ *respectively.* ■

Another change over the base algorithm is that we cache the pairs already returned to the user in an unsorted list for later processing ($queue_{old}$).

For joins, the distance function can change in its eccentricity and orientation in space. Figure 8.6 shows iso-distance contours of the same distance for an original and refined distance function.[8] All data point and node pairs in the region of the original query have been explored and are included in *queue* or $queue_{old}$. All possible point or node pairs in the shaded region can be re-utilized for computing the result of the refined query. In the figure, point $P2$ was returned for this query point in the previous iteration and is also included in the new iteration. $P3$ however is in a region that has not been explored in connection with the present query point, and so may incur additional disk accesses.

The algorithm in table 8.6 improves the efficiency for subsequent query iterations by caching

[8]This figure shows distance functions overlapped with a single data point from the query dataset, these functions are conceptually overlapped with all data points in the query dataset and compared to the other dataset to find the nearest pairs of points, as depicted in figure 8.7.

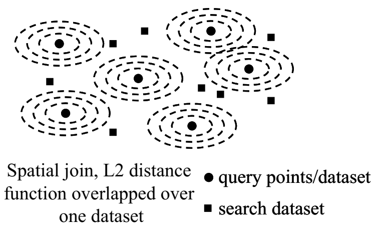

Spatial join, L2 distance function overlapped over one dataset
● query points/dataset
■ search dataset

Figure 8.7: Join Distance Function

the priority queue and results from earlier iterations.[9] After a user is done viewing the results of a query iteration, submits feedback and the system computes a new distance function, we initialize a new iteration by constructing a new priority queue $queue'$. We construct the new priority queue $queue'$ by recomputing the distance for each pair from the original priority queue $queue$ and the data already returned to the user $queue_{old}$. This ensures that we have updated the algorithms state for the new distance function and preserve the correctness property so we can continue exploring at will. Note that if the list $queue_{old}$ becomes too large to remain in memory, it can be sequentially written to disk and later sequentially read to be included in $queue'$. After all items are transferred, $queue_{old}$, and $queue$ are discarded and $queue'$ becomes $queue$ ($queue \leftarrow queue'$). This process repeats for subsequent iterations.

8.6 Evaluation

We conducted an extensive empirical study to (1) evaluate the multipoint approach to answering multipoint k-NN selection queries and compare it to the multiple expansion approach, (2) evaluate the proposed k-NN selection query techniques, namely, full reconstruction (FR) and selective reconstruction (SR), to answering refined queries over a single feature and compare them to the naive approach, and (3) evaluate the proposed top-k join query techniques to answering refined queries. We conducted our experiments on real-life multimedia and spatial datasets. The major findings of our study are:

- **Efficiency of multipoint approach:** The k-NN search based on our multipoint approach is more efficient than the multiple expansion approach. The cost of the latter increases linearly with the number of points in the multipoint query while that of the former is independent of the number of query points.

- **Speedup obtained for refined queries:** The FR and SR approaches speed up the execution of refined queries by almost two orders of magnitude over the naive approach. As expected, the SR approach is the most efficient among all approaches. The sequential scan is significantly slower than any of the index based approaches. Likewise, the speedup for refined

[9]We implemented the pruning option from [72] (cf. sec. 8.2.4.2), we do shown it here for clarity reasons.

```
type { ($node$ | $point$) : A, ($node$ | $point$) : B} : pair
variable $queue$: MinPriorityQueue(distance, pair);
variable $queue_{old}$: Queue(distance, pair);

function NewIteration()
1. /* $\mathcal{D}_Q$, MINDIST, and MINDIST_RECT were */
2. /* modified by the feedback process of section 7.4 */
3. variable queue': MinPriorityQueue(distance, pair);
4. while not $queue_{old}$.empty() do /* process earlier returns */
5.     top = $queue_{old}$.pop()
6.     /* top.A and top.B must be points */
7.     $queue'$.insert($\mathcal{D}_Q(top.A, top.B)$, $\langle top.A, top.B \rangle$)
8. enddo
9. while not $queue$.empty() do
10.    top = $queue$.pop()
11.    if top.A and top.B are points
12.        $queue'$.insert($\mathcal{D}_Q(top.A, top.B)$, $\langle top.A, top.B \rangle$)
13.    else if top.A and top.B are nodes
14.        $queue'$.insert($MINDIST\_RECT(top.A, top.B)$, $\langle top.A, top.B \rangle$)
15.    else if top.A is a node and top.B is a point
16.        $queue'$.insert($MINDIST(top.B, top.A)$, $\langle top.A, top.B \rangle$)
17.    else if top.A is a point and top.B is a node
18.        $queue'$.insert($MINDIST(top.A, top.B)$, $\langle top.A, top.B \rangle$)
19.    endif
20.enddo
21.$queue = queue'$ /* the re-processed queue becomes $queue$ */
end function

function GetNextPair()
1. while not $queue$.IsEmpty() do
2.     top=$queue$.pop();
3.     if top.A and top.B are objects /*i.e., points*/
4.         $queue_{old}$.append(top); // keep it in history
5.         return top;
6.     else if top.A and top.B are nodes
7.         for each child $o_1$ in top.A
8.             for each child $o_2$ in top.B
9.                 if $o_1$ and $o_2$ are nodes
10.                    $queue$.insert($MINDIST\_RECT(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
11.                if $o_1$ is a node and $o_2$ is a point
12.                    $queue$.insert($MINDIST(o_2, o_1)$, $\langle o_1, o_2 \rangle$);
13.                if $o_1$ is a point and $o_2$ is a node
14.                    $queue$.insert($MINDIST(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
15.                if $o_1$ is a node and $o_2$ is a point
16.                    $queue$.insert($\mathcal{D}_Q(o_1, o_2)$, $\langle o_1, o_2 \rangle$);
17.    else if top.A is a node and top.B is a point
18.        for each child $o_1$ in top.A
19.            if $o_1$ is a node
20.                $queue$.insert($MINDIST(top.B, o_1)$, $\langle o_1, top.B \rangle$);
21.            if $o_1$ is a point
22.                $queue$.insert($\mathcal{D}_Q(o_1, top.B)$, $\langle o_1, top.B \rangle$);
23.    else if top.A is a point and top.B is a node
24.        /* mirror of above with top.A and top.B reversed*/
25.    else if top.A is a point and top.B is a point
26.        $queue$.insert($\mathcal{D}_Q(top.A, top.B)$, $\langle top.A, top.B \rangle$);
27.    end if
28.enddo
end function
```

Table 8.6: Incremental Join Algorithm

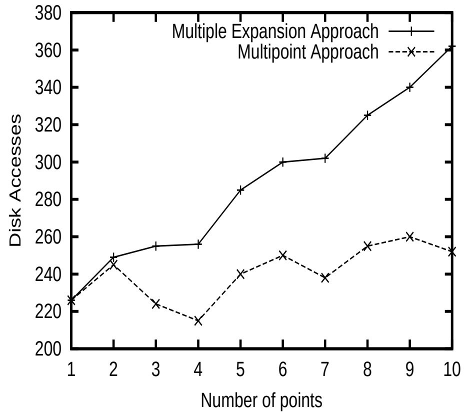


Figure 8.8: I/O Cost of Multipoint Approach and Multiple Expansion Approach

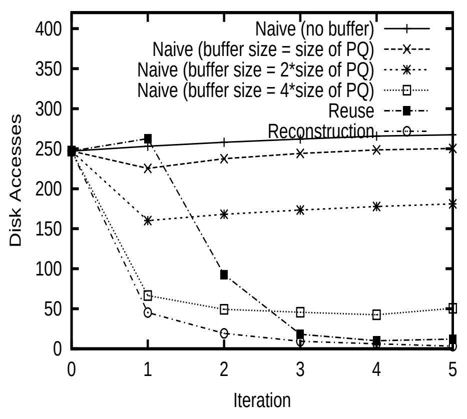


Figure 8.9: I/O Cost of Naive and Reconstruction Approaches for QEX Queries

join queries easily represents an order of magnitude improvement over the naive re-execution approach.

Thus, our experimental results validate the thesis of this chapter that the proposed approaches to evaluating refined queries offer significant speedups over the naive approach. All experiments reported in this section were conducted on a Sun Ultra Enterprise 450 with 1GB of physical memory and several GB of secondary storage, running Solaris 2.7.

In general, the priority queues for the selection and join algorithms may grow too large to remain in main memory. Many works on nearest neighbor retrieval assume that the entire priority queue fits in memory [86, 139].[10] In our NN search experiments, when $k = 100$, the size of priority queue varies between 512KB and 640KB (the size of the index is about 11MB) which can easily fit in memory. Therefore, for our NN search experiments, we assume the priority queue remains in main memory. For the join operation, the priority queue can be significantly larger. The size needed depends on the dimensionality of the data, and on the number of entries which itself depends on the degree of overlap between index nodes. There are several approaches to storing a priority queue on disk including a slotted approach presented in [72], and a collection of sorted lists with guaranteed worst case performance [18]. In our join experiments, even when fetching the top 10,000 pairs out of a possible 1.5 billion, the maximum size of the priority queue was about 1.6MB, well within limits of main memory. Therefore, for our join experiments, we isolated our algorithm from the issue of an external priority queue implementation and assume the priority queue is small enough to fit in memory. A disk based priority queue, although important, is beyond the scope of this chapter.

8.6.1 k-NN Selection Query Evaluation

We first discuss our results for evaluating k-NN selection queries.

[10]This assumption is reasonable when the number k of top matches requested is relatively small compared to the size of the database which is usually the case [21].

8.6.1.1 Methodology

We conducted our experiments for single feature queries on the **COLHIST** dataset comprised of 4x4 color histograms extracted from 70,000 color images obtained from the Corel Database (obtained from `http://corel. digitalriver.com`) [25, 122]. We use the Hybrid tree as the F-index for the 16-dimensional color histograms [25]. We chose the hybrid tree since (1) It is a feature-based index structure (necessary to support arbitrary distance functions)and (2) It scales well to high dimensionalities and large-sized databases [25]. We choose a point Q_C randomly from the dataset and construct a graded set of its top 50 neighbors (based on L_1 distance)[11] i.e., the top 10 answers have the highest grades, the next 10 have slightly lower grades etc. We refer to this set the relevant set $relevant(Q_C)$ of Q_C [122]. We construct the starting query by slightly disturbing Q_C (i.e., by choosing a point close to Q_C) and request for the top 100 answers. We refer to the set of answers returned as the retrieved set $retrieved(Q_C)$. We obtain the refined query Q_C^{new} by taking the graded intersection of $retrieved(Q_C)$ and $relevant(Q_C)$, i.e., if an object O in $retrieved(Q_C)$ is present in $relevant(Q_C)$, it is added to the multipoint query Q_C^{new} with its grade in $relevant(Q_C)$. The goal here is to get $retrieved(Q_C)$ as close as possible to $relevant(Q_C)$ over a small number of refinement iterations. The intra-feature weights were calculated using the techniques described in [122]. In all the experiments, we perform 5 feedback iterations in addition to the starting query (counted as iteration 0). All the measurements are averaged over 100 queries. In our experiments, we fix the hybrid tree page size to 4KB (resulting in a tree with 2766 nodes).

8.6.1.2 Multipoint Query Results

We compare the multipoint approach to evaluating single feature k-NN queries to the multiple expansion approach proposed in FALCON [169] and MARS [125]. Figure 8.8 compares the cost of the two approaches in terms of the number of disk accesses required to return the top 100 answers (no buffering used). The I/O cost of the multipoint approach is almost independent of the number of points in the multipoint query while that of the multiple expansion approach increases linearly with the number of query points. The reason is that since the multiple expansion approach explores the neighborhood of each query point individually, it needs to see more and more unnecessary neighbors as the number of query points increases.

8.6.1.3 Query Expansion Results

We first present the results for the QEX model. Figure 8.9 compares the I/O cost of the naive and reconstruction approaches for the QEX model. In this experiment, the size of cached priority queue varies between 512KB and 640KB (across the iterations). We first run the naive approach with no buffer. We also run the naive approach with an LRU buffer for three different buffer

[11]We use the L_1 metric (i.e., Manhattan distance) as the distance function $\mathcal{D}_Q$ for the color feature since it corresponds to the histogram intersection similarity measure, the most commonly used similarity measure for color histograms [114, 115].

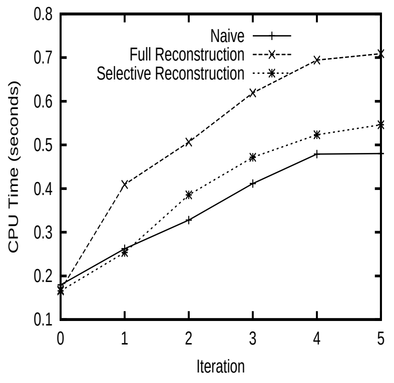


Figure 8.10: CPU Time of Full and Selective Reconstruction Approaches for QEX

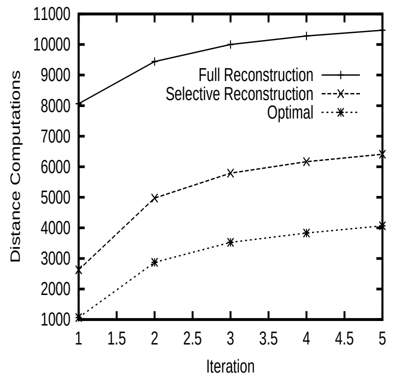


Figure 8.11: Distance Computations saved by Stopping Condition in SR Approach

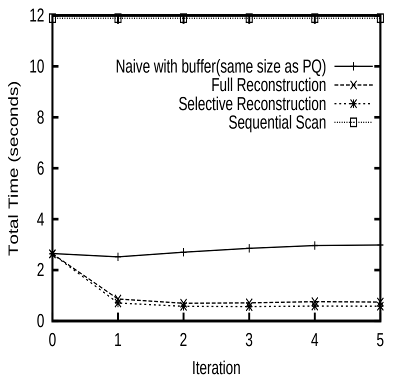


Figure 8.12: Response Time of Naive, Full and Selective Reconstruction Approaches for QEX

sizes: 540KB, 1080KB and 2160KB, i.e., they would hold 135, 270 and 540 of the most recently accessed nodes (4KB each) of the F-index (which has 2766 nodes) respectively. Note that the above 3 buffers use approximately the same amount of memory as the reconstruction approach (to keep the priority queue cached in memory), twice as much and four times as much respectively. We also implemented a type of session aware cache we call the *Reuse* approach, which is a buffer organized as the priority queue itself. When a new iteration starts, we push back all the returned items into the queue without any modification, this has the effect of caching all the information seen so far. Note that since at each iteration we start with all items seen in the previous iteration in the queue, this algorithm is not IO optimal as it may explore nodes again with the new distance function. This buffer is then traversed based on a LBD as in the SR approach, we used the optimal LBD for this experiment by exploring the entire queue to reflect the best possible use of this buffering approach. We implemented this session aware buffering approach to show that even with intelligent buffering our approach is superior.

The reconstruction approach (with no additional buffer besides the priority queue cache) significantly outperforms the naive approach, even when the latter uses much larger buffer sizes (up to 4 times more). While the reconstruction approach is I/O optimal (i.e., accesses a node from the disk at most once during the execution of a query), the naive buffer approach, given the same amount of memory, needs to access the same nodes multiple times from disk across the refinement iterations of the query. In more realistic environments where multiple query sessions belonging to different users run concurrently and users have "think time" between iterations of feedback during a session, we expect the buffer hit rate to drop even further, causing the naive approach to perform even more repeated disk accesses and making our approach of per-query caching even more attractive [61]. The reuse approach performs better than any of the LRU based approaches after the second iteration (remember that the reuse approach is not IO optimal), but worse than our reconstruction approach. Even after five iterations, the reuse approach performs roughly three times as many

IOs than our reconstruction approach. Figure 8.10 compares the CPU cost of the FR and SR approaches for the QEX model. The SR approach significantly outperforms the FR approach in terms of CPU time. Figure 8.11 compares the number of distance computations performed by the FR approach, the SR approach with the proposed stopping condition and the SR approach with the optimal stopping condition (range P2 in figure 8.3). The proposed stopping condition saves more than 50 % of the distance computations performed by FR while optimal stopping condition would have saved about 66 % of the distance computations. This shows that the proposed stopping condition is quite close to the optimal one. Figure 8.12 compares the average response time (sum of I/O wait time and CPU time) of a query for the naive (with buffer), FR, SR and sequential scan approaches (assuming that the wait time of a random disk access is 10ms and that of a sequential disk access is 1ms [62]). The SR technique outperforms all the other techniques; it outperforms the naive approach by almost an order of magnitude over the naive approach and the sequential scan approach by almost two orders of magnitude.

8.6.1.4 Query Point Movement Results

Figure 8.13 compares the I/O cost of the our reconstruction approach and the same reuse buffering approach with optimal LBD used in section 8.6.1.3 (no additional buffer in either case) for the QPM model. Again, the I/O optimal reconstruction approach is far better compared the the reuse approach. Figure 8.14 compares the SR and FR approaches with respect to the CPU cost. Unlike in QEX approach where the SR approach is significantly better than the FR approach, in QPM the former is only marginally better. The reason is that, unlike in the QEX approach where the savings in distance computations far outweighs the cost of 'push back', the savings only marginally outweighs the extra cost in the case of QPM. This is evident in figure 8.15 which shows that the the number of distance computations performed by the SR approach is much closer to that performed by the FR approach as compared to the QEX model. Even the optimal stopping condition would not save as many distance computations as the QEX model. This is due to dimension reweighting that causes the stopping condition to become more conservative (by introducing the factor K in Lemma 8.4.2) resulting in less savings in distance computations. However, with a more efficient 'push back' strategy, we expect the SR approach to be significantly better than the FR approach in this case as well.

SR improves response time over FR for the top few matches. In an interactive user environment, it is important to consider not only the time required to compute the top k answers, but how soon they can be produced. Figure 8.16 compares FR and SR for the QPM model in terms of the relative response time for the first answer out of top 100 queries. In FR, while the queue is transferred, no answers are produced. By contrast, SR will produce the first answer as soon as it is possible, therefore producing faster the answers users see first. The figure plots the fraction of time SR needs to produce the top answer relative to FR which is always 100%.

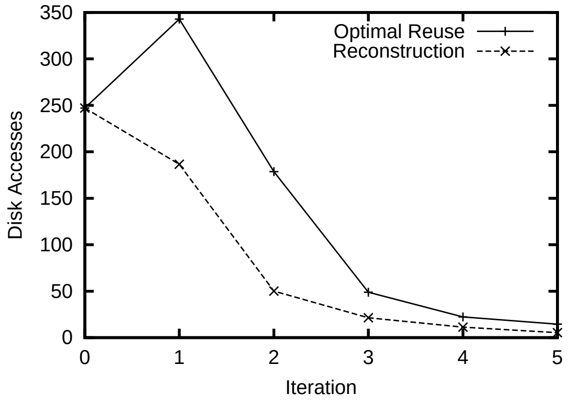


Figure 8.13: I/O Cost of Reuse and Reconstruction Approaches for QPM

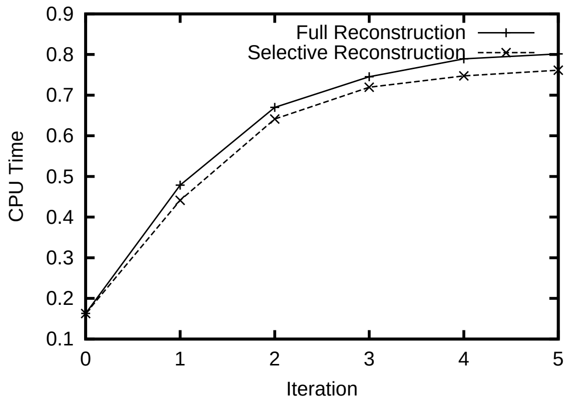


Figure 8.14: CPU Time for Full and Selective Reconstruction Approaches for QPM

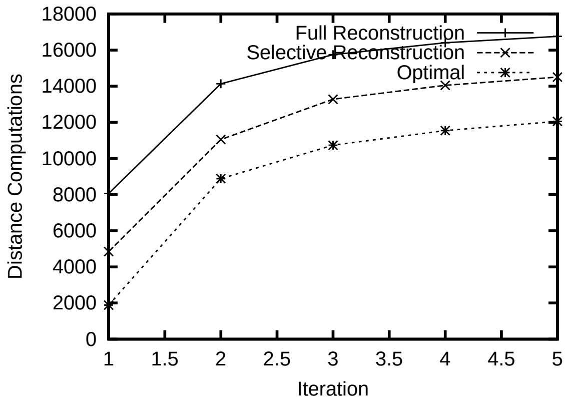


Figure 8.15: Distance computations saved by Stopping Condition for QPM

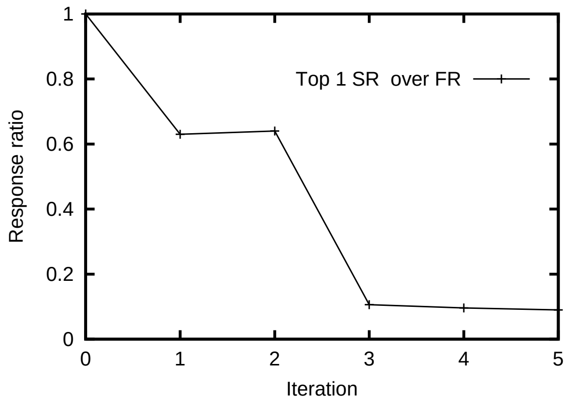


Figure 8.16: Response Ratio SR/FR for the top 1 Item in a KNN Query

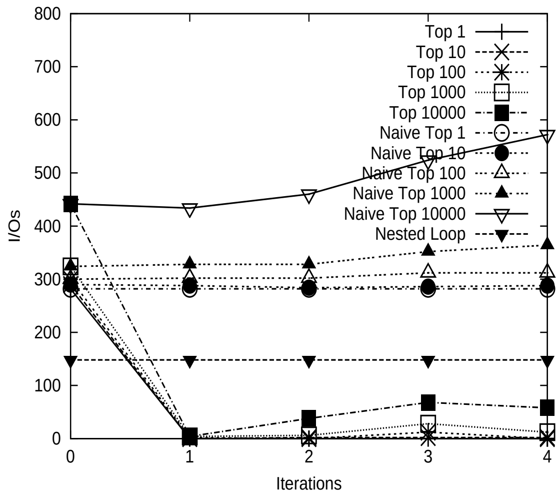


Figure 8.17: I/O cost of naive, reconstruction, and nested loop approaches

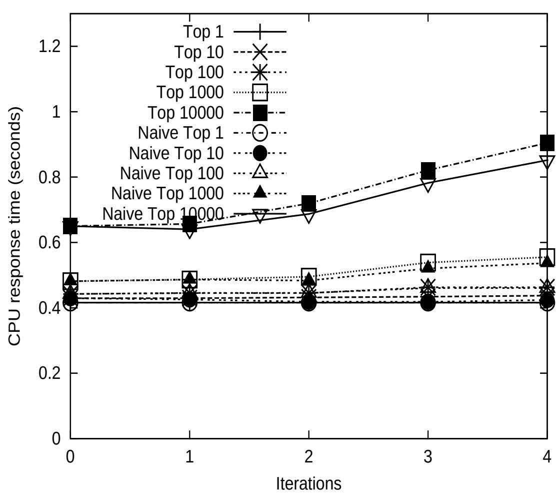


Figure 8.18: CPU time of naive and reconstruction approaches

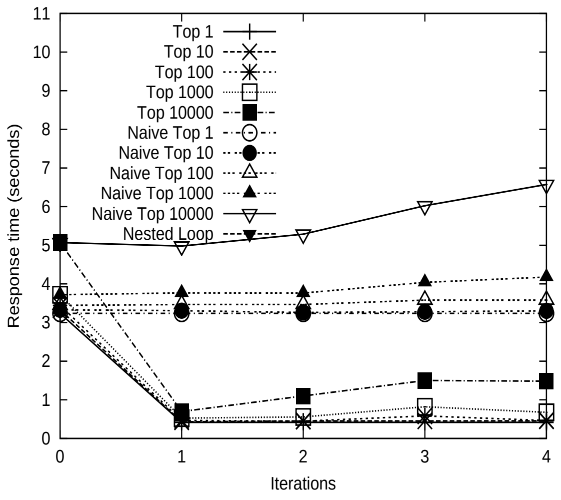


Figure 8.19: Total response time of naive and reconstruction approaches

8.6.2 *k*-NN Join Query Evaluation

We now turn our attention to evaluating the performance of join queries. For this experiment, we chose a two dimensional join based on geographic data which could simulate the geographic distance join of example 1.0.2.

8.6.2.1 Methodology

For these experiments, we used the following two datasets: (1) the fixed source air pollution dataset from the AIRS[12] system of the EPA which contains 51,801 tuples with geographic location and emissions of 7 pollutants (carbon monoxide, nitrogen oxide, particulate less than 2.5 and 10 micrometers,

[12]http://www.epa.gov/airs

sulfur dioxide, ammonia, and volatile organic compounds), (2) the US census data which contains 29470 tuples that include the geographic location at the granularity of one zip code, population, average and median household income. We constructed an R*-tree for each of these datasets with a 2KB page size, for a maximum fan out of 85. The pollution dataset index had 923 nodes and the census index had 548 nodes.

We focused only on the geographic location attribute and constructed queries with different eccentricities and orientations. Then, we used these queries as starting and goal queries and let the query refinement generate the intermediate queries. We tested starting to goal query difference in eccentricity of up to 2 orders of magnitude, which represents a substantial change in the distance function, far more than can be expected under normal query circumstances. Even so, our algorithm performed quite well.

For the same sets of starting and target queries, we ran tests to obtain the top 1, 10, 100, 1000, and 10,000 pairs[13] that match the query and performed feedback on them.[14] For each of these, we measured the number of disk I/Os performed, the cpu response time (ignoring the priority queue overhead), and the wall clock response time (subtracting the priority queue overhead).

8.6.2.2 Results

Figure 8.17 shows the number of node accesses performed (assuming no buffering) by the naive approach (re-executing the query from scratch every time) and our reconstruction approach. After the initial query, subsequent iterations perform minimal node accesses for the reconstruction approach while the naive approach roughly performs the same work at each iteration. We also compare to the nested loop join algorithm. Since we assume the priority queue for our algorithm has unlimited buffer space in memory, to be fair to nested loop join, we assume unlimited memory and fit both complete datasets in memory (thus only 1 access per page). We included the disk accesses needed by the nested loop join algorithm and divided this value by 10 to reflect the relative advantage of sequential reads over random reads [62]. Thus, the roughly 150 accesses already take into account the advantages gained from sequential access. While slightly lower than the naive re-execution, nested loop suffers from a very high cpu overhead and easily loses out to the naive algorithm when overall time is considered (2-3 orders of magnitude). Figure 8.18 shows the CPU time needed by the algorithm excluding the priority queue overhead. The time required for the nested loop join computations (ignoring the sort time since we ignored the priority queue overhead above) is roughly 21 minutes for the approximately 1.5 billion possible pairs. The cpu time includes the cost of reconstructing the priority queue for each new iteration. Here, due to accumulated information in the queue, the reconstruction approach has a slight disadvantage over the naive approach, it performs about 5-10% slower than the naive approach. Figure 8.19 shows the total wall clock time

[13]We ran these tests independently, i.e., information in the priority queue was not shared among them.

[14]For the experiment that requests only the top pair we did not use feedback, since there is not enough information to meaningfully compute a new distance function. Instead we used the per-iteration distance functions of the top 100 query.

(excluding priority queue overhead) for the queries. Nested loop join (ignoring sort time) takes roughly 22 minutes to complete this query even when both datasets fit entirely in memory. This is due to the very high number of distance computations required. The total response time using the reconstruction approach is typically 3-4 times faster than using the naive approach. Overall, the reconstruction technique significantly outperforms the naive approach or the nested loop join technique.

8.7 Conclusion

Top-k selection queries are becoming common in many modern-day database applications like multimedia retrieval and e-commerce applications. In such queries, the user specifies target values for certain attributes and expects the "top k" objects that best match the specified values. Due to the user subjectivity involved in such queries, the answers returned by the system often do not satisfy the user's need right away. In such cases, the system allows the user to refine the query and get back a better set of answers. Despite much research on query refinement models, there is no work that we are aware of on supporting refinement of top-k queries efficiently in a database system. Done naively, each 'refined' query is treated as a 'starting' query and evaluated from scratch. We propose several techniques to evaluate refined queries *efficiently*. Our techniques save most of the execution cost of the refined queries by appropriately exploiting the information generated during the previous iterations of the query. Our experiments show that the proposed techniques provide significant improvement over the naive approach in terms of the execution cost of refined queries.

Chapter 9

Case Study

9.1 Introduction

To explore the usefulness and feasibility of our work in real world applications, we set out to construct two catalog based applications. The first application is a catalog based clothing store where users can search for clothes based on images of garments, price, description, etc. As users explore the store, they provide feedback that guides the query and results in improved answers. The second application we explore is that of online car shopping. Users can search a variety of used cars using multiple criteria and interactively refine their queries to improve their results. We also implemented this second application in a commercial database system to contrast both, development time and system performance. The results are very encouraging.

9.2 Garment Search

In this case study, we show a possible application for clothing retrieval where the price, description and image content are used to search garments. We first give an overview of the features used and then show an interaction with our prototype system and some example screen shots of the equivalent interaction.

We built a Java based front end user interface for the garment search application that lets users browse, construct queries and refine answers. Users construct queries by selecting one or more values for each attribute (single– or multi–point similarity predicate). Values are provided either directly by typing in the value, or indirectly, by selecting example attribute values while browsing the collection. The image attribute is the only one for which selecting examples is the only choice since searching is done on the color and texture feature values and it is virtually impossible for users to type in sensible values for these attributes. The user interface then generates a SQL statement which is sent to our prototype for querying and displays the best answers in rank order. We display 10 answers at a time and let the user scroll through multiple screens. While viewing answers, users can indicate that the entire answer is relevant through a checkbox at the left side of the tuple, or give detailed column level feedback by exploring the item in detail. Feedback on the image is converted to feedback on the image feature attributes.

9.2.1 Features Used

The retrieval performance of any similarity matching system is inherently limited by the nature and the quality of the features used to represent the objects content. Features representations for various domains such as image retrieval, time-series retrieval, video retrieval and others are a very active research area. This section gives a brief overview of the content based features we use for our case study.

9.2.1.1 Image Features

We make use of the color, texture and shape features. These are described in detail in appendix A. For the image features we use the intra-predicate feedback algorithms discussed in chapter 7.

9.2.1.2 Text Features

IR technology is directly applicable to text-based retrieval for multimedia objects. Objects in IR are referred to as documents. Each document is viewed as a very high multidimensional vector where each word (*term*) represents a dimension. Note that commonly used words – such as, is, has, I, he – are ignored since they are not useful for retrieval (*stop list*). Let w_{ij} (*term weight*) be the value in dimension j of the vector representation for document i. w_{ij} is computed as $tf \times idf$ where tf (*term frequency*) of w_{ij} is the frequency of term j in document i and idf (*inverse document frequency*) for term j is computed as $\log \frac{N}{n}$ where N is the total number of documents in the collection and n is the number of documents that contain term j. Finally, each vector is normalized to a length of 1 which simplifies similarity computations; i.e., $w_{ij} = w_{ij}/\sqrt{\sum_{j=1}^{m} w_{ij}^2}$ where m is the total number of terms (dimensions) in the document space, see [168] for more optimizations of text matching. *Cosine Similarity* (cosine) is used to compute the similarity between two documents (Doc_i and Doc_q) [137].

$$\cos(Doc_i, Doc_q) = \sum_{j=1}^{m} w_{ij}\ w_{qj} \tag{9.1}$$

Similar to the relationship between the histogram intersection and the Manhattan distance, the cosine similarity is a function of the Euclidean distance (L_2-distance) as shown in (9.2).

$$\cos(D_i, D_q) = 1 - \frac{(L_2(Doc_i, Doc_q))^2}{2} \qquad \text{where} \qquad L_2(D_i, D_q) = \sqrt{\sum_{j=1}^{m} (\ w_{ij} - w_{qj}\ )^2} \tag{9.2}$$

For refinement, we use the query point movement approach presented in chapter 7.

9.2.2 Using Our System

Now that we know the features used, we turn to explore how they can be used in our prototype database system.

First, we create the types for each non-built in type:

```
create type ImageThumbnail (an_im blob) functions "types_dll/types.so:create(13)"
create type ColorHistogram float(32) functions "types_dll/types.so:create(14)"
create type CoocTexture float(16) functions "types_dll/types.so:create(16)"
create type norfloat (x float) functions "types_dll/types.so:create(12)"
create type varcharindex (x varchar) functions "types_dll/types.so:create(8)"
```

For each type, its structure is specified followed by a specification of a dynamic library in which to find the functions that implement the type (see section 3.2). The *ImageThumbnail* is a structure with a single data value of type blob and is used to store a reduced version of an image, a thumbnail. *ColorHistogram* and *CoocTexture* are floating point number arrays and their features work as described above in section 9.2.1. *Norfloat* is a single floating point number, but it is normalized as a Gaussian sequence, each time a number is inserted, sufficient meta-data is kept to perform the necessary normalization at query time. The *varcharindex* type is based on the *varchar* character string type but augments it with the desired similarity functions described above in section 9.2.1.

Now, we create a table to store our data:

```
create table esearch(
    gender char,
    manuf varcharindex,
    short_desc varcharindex,
    price norfloat,
    thumbnail ImageThumbnail,
    histogram ColorHistogram,
    texture CoocTexture)
fragment (2) with extent (200,200)
tid (2) with extent (200,200)
put "thumbnail.an_im" into (2) with extent (200,200)
functions "tables_dll/tables.so:create_table_interface(1)"
```

This table has columns for our garment data, with the gender as a single character, the manufacturer and a short description are text fields that are enabled for similarity searching, price is a normalized number, and finally the image data: thumbnail, color histogram and texture. The rest of the table definition indicates to stripe the table and the tid index over disk 2 only and uses 200 pages as the primary and secondary extents. The blob in the thumbnail is also stored in disk 2. The final specification is of the table functions needed, which in this case implement the standard table interfaces.

Before we can use our new table we need to specify some helpful indices:

```
create index esearch_manuf_idx
       table esearch(manuf)
       using text_vector_model(0)
       fragment (2) with extent (50,50)
create index esearch_short_desc_idx
       table esearch(short_desc)
       using text_vector_model(0)
       fragment (2) with extent (50,50)
```

These indices over the text fields support the above described text vector model and are also stored on disk 2 with primary and secondary extent sizes of 50.

Now, we are ready to insert some data:

```
insert into esearch values ("M", "Eddie Bauer", "Fleece Jacket",
                            "59.99", [1], [1], [1]) #file:image.jpg#
```

Since images cannot be readily typed into a textual interface, they are sent as blobs with our client–server library. The image itself is in the file *image.jpg* and is loaded by the client side and transferred as a blob to the server which finds the image for the thumbnail, histogram and texture in the first position of extra blob attributes. Each tuple upon insertion is assigned a unique tuple id (*tid*) that identifies that tuple in the table.

We populated this table with sample data from various apparel web sites[1], we collected 1747 garment item descriptions of a variety of garment types including pants, shirts, jackets, and dresses.

With the datase loaded, we now turn to performing a query:

```
select garments with price, manuf, thumbnail, histogram, texture
       from esearch
       where
         gender = "F"
       and
         price ∼= "80.00" ∼and histogram ∼= (50,60) ∼and texture ∼= (50,60)
```

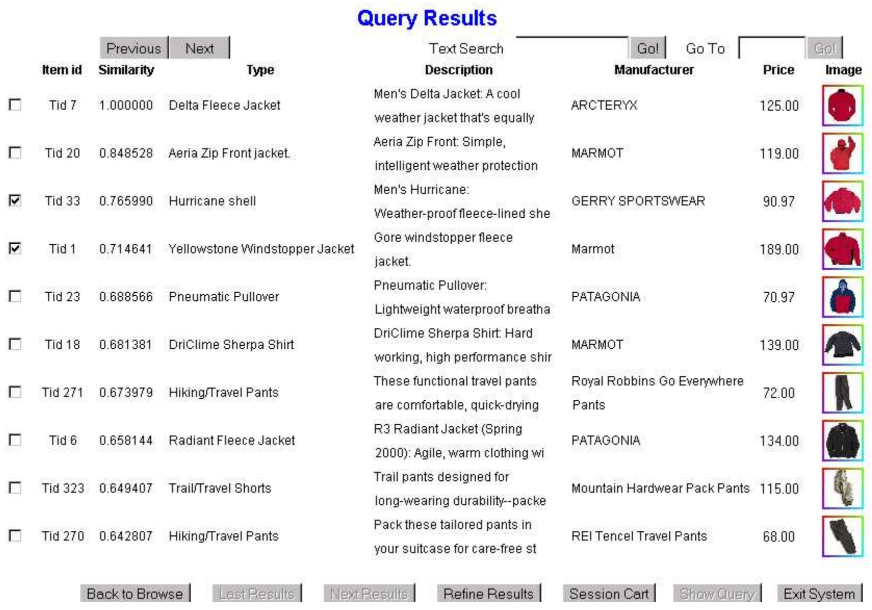
Here, *garments* is the name we have given to our session query and is the handle we can use to submit feedback. The user has indicated he wants female garments and a desired target price of 80 dollars, and wants images that are similar to the color histogram and texture of images 50 and 60 which are stored in the table, thus using a multi-point query. This condition will be separated into the crisp condition $gender$ = "F" and the similarity condition $price \sim=$ "80.00" $\sim$ and $histogram \sim= (50, 60) \sim$ and $texture \sim= (50, 60)$.

Assuming the user has seen the first few results and selects feedback similarly to this table:

tid	price	manufacturer	thumbnail	histogram	texture
415	√	√	–	√	–
319	√	√	√	√	√
128	–	–	–	×	√

To send this information to the database server we use the following:

[1]The sources used are: JCrew, Eddie Bauer, Landsend, Polo, Altrec, Bluefly, and REI.


Query Results

	Item id	Similarity	Type	Description	Manufacturer	Price	Image
☐	Tid 7	1.000000	Delta Fleece Jacket	Men's Delta Jacket: A cool weather jacket that's equally	ARCTERYX	125.00	
☐	Tid 20	0.848528	Aeria Zip Front jacket.	Aeria Zip Front: Simple, intelligent weather protection	MARMOT	119.00	
☑	Tid 33	0.765990	Hurricane shell	Men's Hurricane: Weather-proof fleece-lined she	GERRY SPORTSWEAR	90.97	
☑	Tid 1	0.714641	Yellowstone Windstopper Jacket	Gore windstopper fleece jacket.	Marmot	189.00	
☐	Tid 23	0.688566	Pneumatic Pullover	Pneumatic Pullover: Lightweight waterproof breatha	PATAGONIA	70.97	
☐	Tid 18	0.681381	DriClime Sherpa Shirt	DriClime Sherpa Shirt: Hard working, high performance shir	MARMOT	139.00	
☐	Tid 271	0.673979	Hiking/Travel Pants	These functional travel pants are comfortable, quick-drying	Royal Robbins Go Everywhere Pants	72.00	
☐	Tid 6	0.658144	Radiant Fleece Jacket	R3 Radiant Jacket (Spring 2000): Agile, warm clothing wi	PATAGONIA	134.00	
☐	Tid 323	0.649407	Trail/Travel Shorts	Trail pants designed for long-wearing durability--packe	Mountain Hardwear Pack Pants	115.00	
☐	Tid 270	0.642807	Hiking/Travel Pants	Pack these tailored pants in your suitcase for care-free st	REI Tencel Travel Pants	68.00	

Figure 9.1: Query Results

```
insert feedback into garments tid = 415 feedback=0 values (1, 1, 0, 3, 0)
insert feedback into garments tid = 319 feedback=1 values (1, 1, 1, 1, 1)
insert feedback into garments tid = 128 feedback=0 values (0, 0, 0, -2, 2)
```

This gives the illusion of a feedback table with the same number of columns as the result. The user can then submit feedback on a tuple level by using the *feedback=X* value and/or individually in a column by column way. Here, for the tuple 415 the user indicates that the price and the manufacturer are in fact desirable and the color is very desirable. For tuple 319, all values are good, and for tuple 128, the color is not at all interesting, but the texture is very nice, perhaps it is a striped pattern.

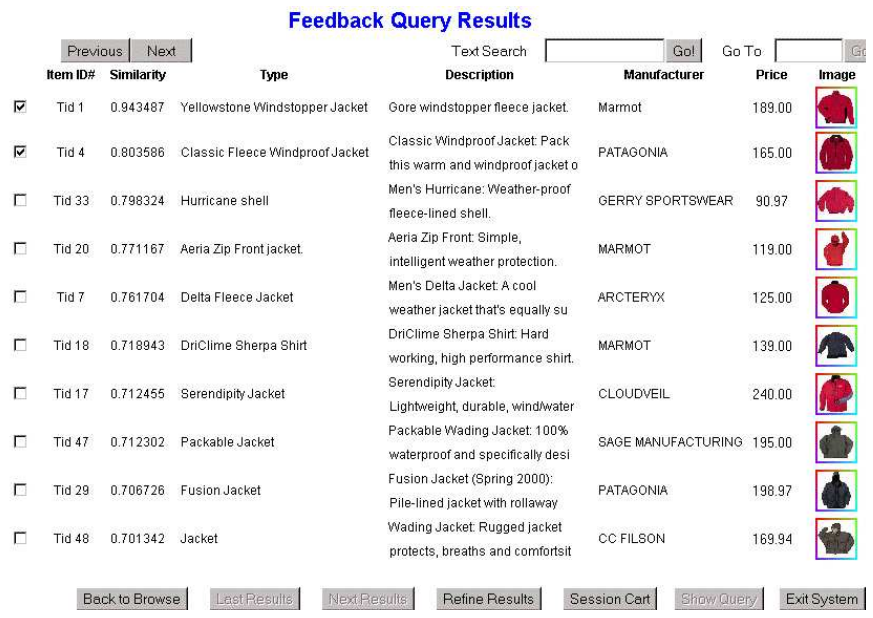
Once the user has submitted the feedback, we refine the query and obtain new results:

```
select garments feedback
```

This is similar to the select statement above, but now we do not specify the table or tables involved nor the conditions. In response to the user supplied information, the query has been changed and the results incrementally modified. The *garments* session query identifier represents to the database server the session query or iterative query that is meant.

Now, the user can continue her browsing of the result, submit more feedback and refine the query until done.

An example user interface that uses these commands was developed and is shown in figures 9.1, 9.2, and 9.3. Figure 9.1 shows the original results for a query and two tuples marked for feedback, then figure 9.2 shows the results of the first feedback iteration and also shows some tuples marked for feedback, the application of which results in the answers shown in figure 9.3.



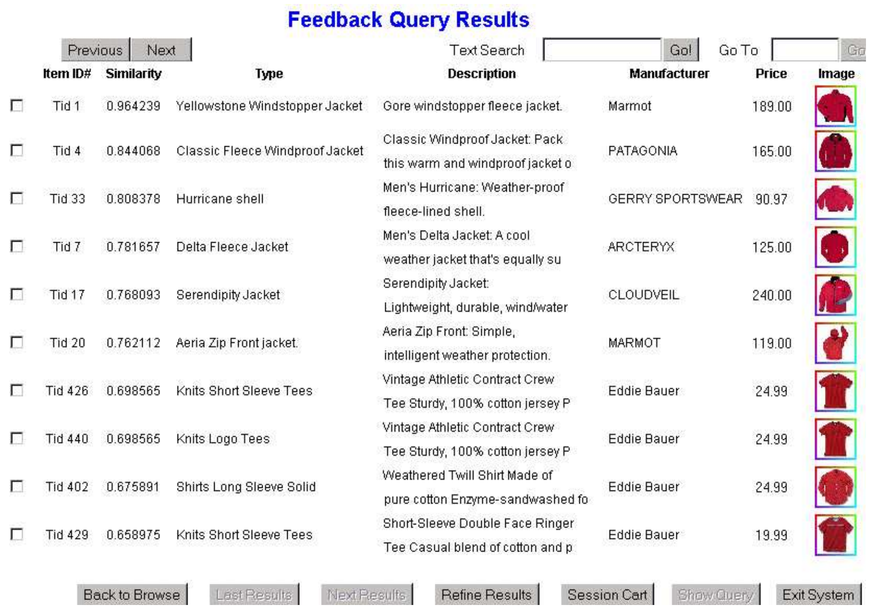
Feedback Query Results

Previous | Next   Text Search [ ] Go!   Go To [ ] Go

	Item ID#	Similarity	Type	Description	Manufacturer	Price	Image
☑	Tid 1	0.943487	Yellowstone Windstopper Jacket	Gore windstopper fleece jacket.	Marmot	189.00	
☑	Tid 4	0.803586	Classic Fleece Windproof Jacket	Classic Windproof Jacket: Pack this warm and windproof jacket o	PATAGONIA	165.00	
☐	Tid 33	0.798324	Hurricane shell	Men's Hurricane: Weather-proof fleece-lined shell.	GERRY SPORTSWEAR	90.97	
☐	Tid 20	0.771167	Aeria Zip Front jacket.	Aeria Zip Front: Simple, intelligent weather protection.	MARMOT	119.00	
☐	Tid 7	0.761704	Delta Fleece Jacket	Men's Delta Jacket: A cool weather jacket that's equally su	ARCTERYX	125.00	
☐	Tid 18	0.718943	DriClime Sherpa Shirt	DriClime Sherpa Shirt: Hard working, high performance shirt.	MARMOT	139.00	
☐	Tid 17	0.712455	Serendipity Jacket	Serendipity Jacket: Lightweight, durable, wind/water	CLOUDVEIL	240.00	
☐	Tid 47	0.712302	Packable Jacket	Packable Wading Jacket: 100% waterproof and specifically desi	SAGE MANUFACTURING	195.00	
☐	Tid 29	0.706726	Fusion Jacket	Fusion Jacket (Spring 2000): Pile-lined jacket with rollaway	PATAGONIA	198.97	
☐	Tid 48	0.701342	Jacket	Wading Jacket: Rugged jacket protects, breaths and comfortsit	CC FILSON	169.94	

Back to Browse | Last Results | Next Results | Refine Results | Session Cart | Show Query | Exit System

Figure 9.2: First Feedback Iteration Results



Feedback Query Results

Previous | Next   Text Search [ ] Go!   Go To [ ] Go

	Item ID#	Similarity	Type	Description	Manufacturer	Price	Image
☐	Tid 1	0.964239	Yellowstone Windstopper Jacket	Gore windstopper fleece jacket.	Marmot	189.00	
☐	Tid 4	0.844068	Classic Fleece Windproof Jacket	Classic Windproof Jacket: Pack this warm and windproof jacket o	PATAGONIA	165.00	
☐	Tid 33	0.808378	Hurricane shell	Men's Hurricane: Weather-proof fleece-lined shell.	GERRY SPORTSWEAR	90.97	
☐	Tid 7	0.781657	Delta Fleece Jacket	Men's Delta Jacket: A cool weather jacket that's equally su	ARCTERYX	125.00	
☐	Tid 17	0.768093	Serendipity Jacket	Serendipity Jacket: Lightweight, durable, wind/water	CLOUDVEIL	240.00	
☐	Tid 20	0.762112	Aeria Zip Front jacket.	Aeria Zip Front: Simple, intelligent weather protection.	MARMOT	119.00	
☐	Tid 426	0.698565	Knits Short Sleeve Tees	Vintage Athletic Contract Crew Tee Sturdy, 100% cotton jersey P	Eddie Bauer	24.99	
☐	Tid 440	0.698565	Knits Logo Tees	Vintage Athletic Contract Crew Tee Sturdy, 100% cotton jersey P	Eddie Bauer	24.99	
☐	Tid 402	0.675891	Shirts Long Sleeve Solid	Weathered Twill Shirt Made of pure cotton Enzyme-sandwashed fo	Eddie Bauer	24.99	
☐	Tid 429	0.658975	Knits Short Sleeve Tees	Short-Sleeve Double Face Ringer Tee Casual blend of cotton and p	Eddie Bauer	19.99	

Back to Browse | Last Results | Next Results | Refine Results | Session Cart | Show Query | Exit System

Figure 9.3: Second Feedback Iteration Results

9.3 Used Car Search

For this demo, we use a dataset containing information of about 8000 used cars. We obtained this dataset from the web. The original dataset had more than 50 attributes; we selected the following attributes for this demo: car make and model, year, mileage, price, savings, exterior color, class, transmission, options and location (city, state, zip and geographical location (latitude and longitude)). We allow searching based on some attributes; others are just informational in nature. For each searchable attribute, we developed a custom data type that defines the notion of similarity and refinement for that attribute. For example, the similarity function of the make_model data type specifies that a Honda Accord is much more similar to a Toyota Camry than to a Ford Explorer SUV. These data types were then simply plugged into the database engine. It took about 1.5 person weeks to build the data types, including the time to fine-tune the similarity and refinement functions for this application. A simple web-based user interface was developed in an additional 0.5 person weeks. Below we report 2 representative queries and their results. The results demonstrate the effectiveness of our approach for database applications that require flexible searching. We also implemented this application as a wrapper on top of Informix to compare its performance to our system. We implemented the similarity predicates as user defined functions and had to extend them with a threshold to limit the results to the top-k answers.

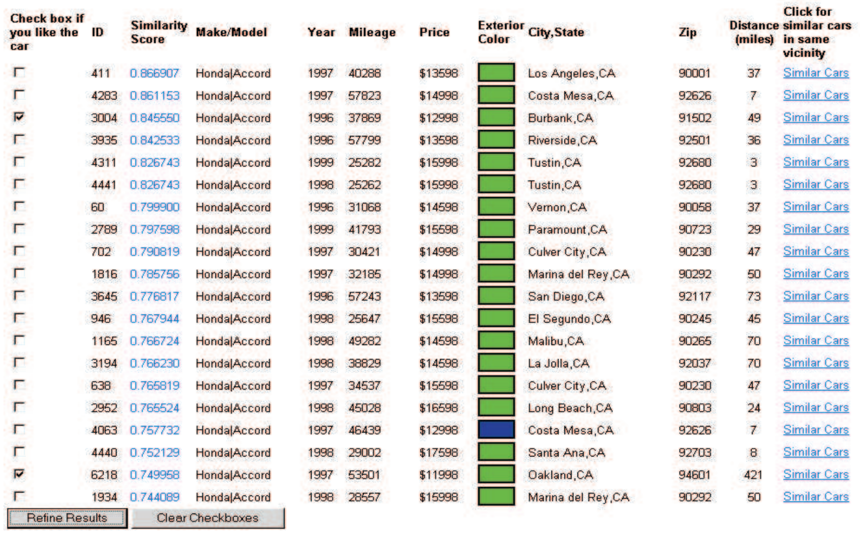
Example 9.3.1 *Example 1: A user Alice is looking for a Honda Accord (around 1997), priced around $12000. She stays in Irvine (zip code 92612). Figure 9.4 is the first screen of results for Alice's query. Alice is conscious about the price but is willing to drive to Bay Area (around 400 miles) to get a good deal. So she marks item 6218 as relevant. Figure 9.5 shows the first screen of refined results.* ■

Example 9.3.2 *Cathy is looking for a pearl colored Acura Integra around Irvine (zip code 92612). Figure 9.6 is the first screen of results for Cathy's query. There is no pearl Acura Integra in the database. The system returns pearl colored cars with similar models (e.g., Honda Accord, Toyota Camry, Nissan Altima etc.) in addition to a silver colored Acura Integra. The silver colored Acura Integra (item 4227) is acceptable to Cathy. Figure 9.7 is the first screen of refined results.* ■

9.3.1 Comparison to a Commercial Database Engine

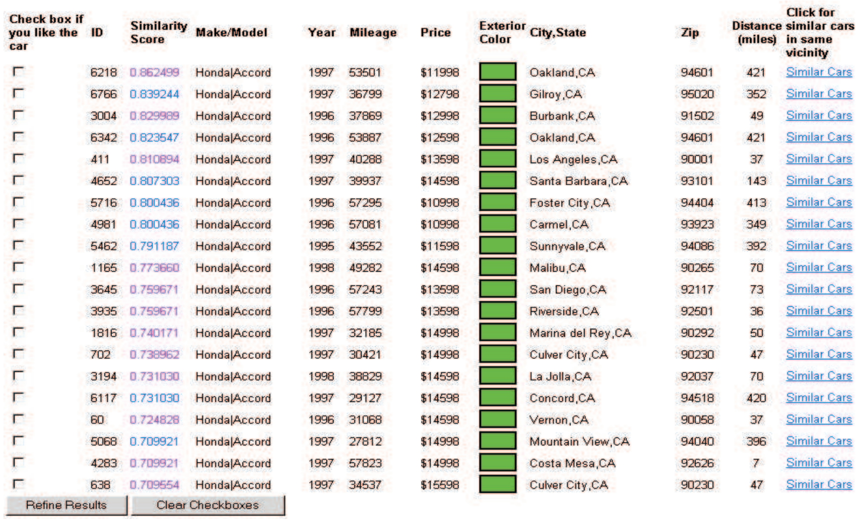
In order to make the comparison, we simulate similarity searching for the automotive application using a straightforward wrapper approach. In the wrapper, the query values for the specified attributes are converted to ranges and the range query is submitted to the database. The retrieved answers are scored and sorted based on the similarity functions before returning them to the user. We used the latest release of the Informix Universal Server for this experiment.

We distinguish ourselves from a wrapper approach using a commercial database in several ways:



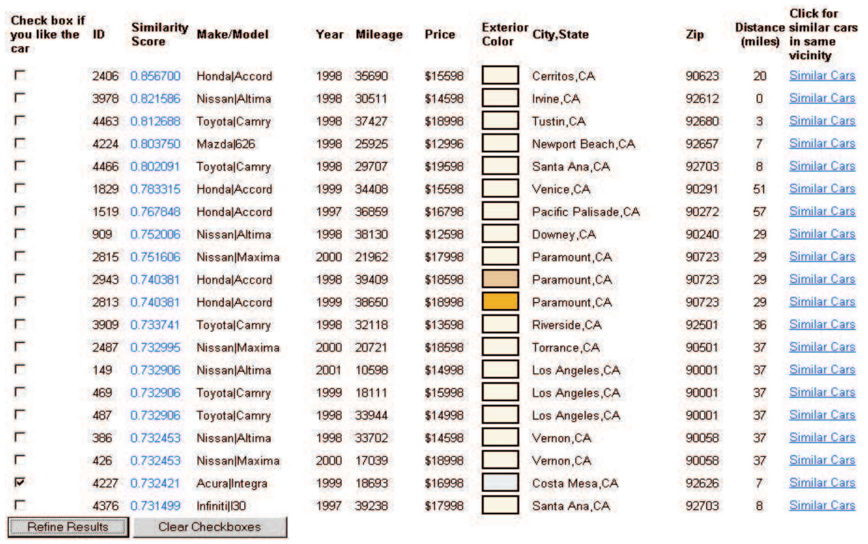
Check box if you like the car	ID	Similarity Score	Make/Model	Year	Mileage	Price	Exterior Color	City,State	Zip	Distance (miles)	Click for similar cars in same vicinity
☐	411	0.866907	Honda|Accord	1997	40288	$13598	🟩	Los Angeles,CA	90001	37	Similar Cars
☐	4283	0.861153	Honda|Accord	1997	57823	$14998	🟩	Costa Mesa,CA	92626	7	Similar Cars
☑	3004	0.845550	Honda|Accord	1996	37869	$12998	🟩	Burbank,CA	91502	49	Similar Cars
☐	3935	0.842533	Honda|Accord	1996	57799	$13598	🟩	Riverside,CA	92501	36	Similar Cars
☐	4311	0.826743	Honda|Accord	1999	25282	$15998	🟩	Tustin,CA	92680	3	Similar Cars
☐	4441	0.826743	Honda|Accord	1998	25262	$15998	🟩	Tustin,CA	92680	3	Similar Cars
☐	60	0.799900	Honda|Accord	1996	31068	$14598	🟩	Vernon,CA	90058	37	Similar Cars
☐	2789	0.797598	Honda|Accord	1999	41793	$15598	🟩	Paramount,CA	90723	29	Similar Cars
☐	702	0.790819	Honda|Accord	1997	30421	$14998	🟩	Culver City,CA	90230	47	Similar Cars
☐	1816	0.785756	Honda|Accord	1997	32185	$14998	🟩	Marina del Rey,CA	90292	50	Similar Cars
☐	3645	0.776817	Honda|Accord	1996	57243	$13598	🟩	San Diego,CA	92117	73	Similar Cars
☐	946	0.767944	Honda|Accord	1998	25647	$15598	🟩	El Segundo,CA	90245	45	Similar Cars
☐	1165	0.766724	Honda|Accord	1998	49282	$14598	🟩	Malibu,CA	90265	70	Similar Cars
☐	3194	0.766230	Honda|Accord	1998	38829	$14598	🟩	La Jolla,CA	92037	70	Similar Cars
☐	638	0.765819	Honda|Accord	1997	34537	$15598	🟩	Culver City,CA	90230	47	Similar Cars
☐	2952	0.765524	Honda|Accord	1998	45028	$16598	🟩	Long Beach,CA	90803	24	Similar Cars
☐	4063	0.757732	Honda|Accord	1997	46439	$12998	🟦	Costa Mesa,CA	92626	7	Similar Cars
☐	4440	0.752129	Honda|Accord	1998	29002	$17598	🟩	Santa Ana,CA	92703	8	Similar Cars
☑	6218	0.749958	Honda|Accord	1997	53501	$11998	🟩	Oakland,CA	94601	421	Similar Cars
☐	1934	0.744089	Honda|Accord	1998	28557	$15998	🟩	Marina del Rey,CA	90292	50	Similar Cars

Figure 9.4: First Screen for Alice



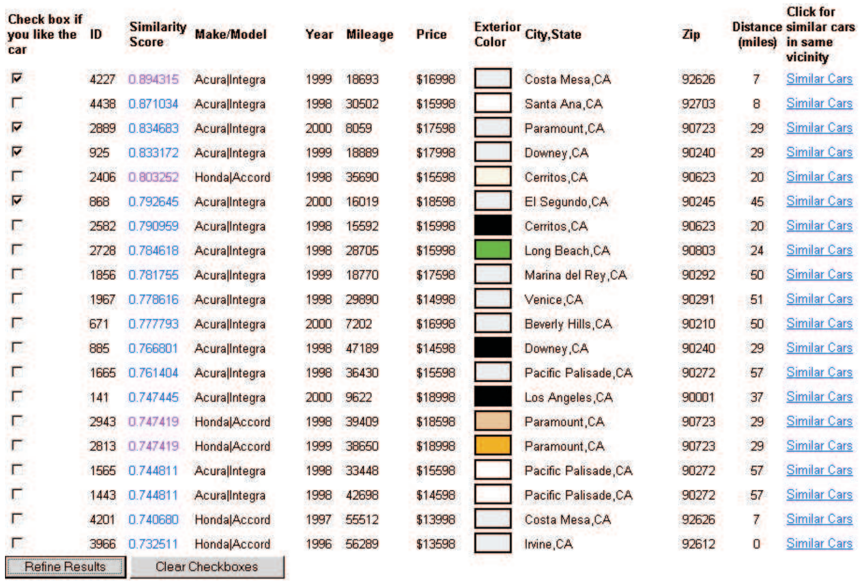
Check box if you like the car	ID	Similarity Score	Make/Model	Year	Mileage	Price	Exterior Color	City,State	Zip	Distance (miles)	Click for similar cars in same vicinity
☐	6218	0.862499	Honda|Accord	1997	53501	$11998	🟩	Oakland,CA	94601	421	Similar Cars
☐	6766	0.839244	Honda|Accord	1997	36799	$12798	🟩	Gilroy,CA	95020	352	Similar Cars
☐	3004	0.829989	Honda|Accord	1996	37869	$12998	🟩	Burbank,CA	91502	49	Similar Cars
☐	6342	0.823547	Honda|Accord	1996	53887	$12598	🟩	Oakland,CA	94601	421	Similar Cars
☐	411	0.810894	Honda|Accord	1997	40288	$13598	🟩	Los Angeles,CA	90001	37	Similar Cars
☐	4652	0.807303	Honda|Accord	1997	39937	$14598	🟩	Santa Barbara,CA	93101	143	Similar Cars
☐	5716	0.800436	Honda|Accord	1996	57295	$10998	🟩	Foster City,CA	94404	413	Similar Cars
☐	4981	0.800436	Honda|Accord	1996	57081	$10998	🟩	Carmel,CA	93923	349	Similar Cars
☐	5462	0.791187	Honda|Accord	1995	43552	$11598	🟩	Sunnyvale,CA	94086	392	Similar Cars
☐	1165	0.773660	Honda|Accord	1998	49282	$14598	🟩	Malibu,CA	90265	70	Similar Cars
☐	3645	0.759671	Honda|Accord	1996	57243	$13598	🟩	San Diego,CA	92117	73	Similar Cars
☐	3935	0.759671	Honda|Accord	1996	57799	$13598	🟩	Riverside,CA	92501	36	Similar Cars
☐	1816	0.740171	Honda|Accord	1997	32185	$14998	🟩	Marina del Rey,CA	90292	50	Similar Cars
☐	702	0.738962	Honda|Accord	1997	30421	$14998	🟩	Culver City,CA	90230	47	Similar Cars
☐	3194	0.731030	Honda|Accord	1998	38829	$14598	🟩	La Jolla,CA	92037	70	Similar Cars
☐	6117	0.731030	Honda|Accord	1997	29127	$14598	🟩	Concord,CA	94518	420	Similar Cars
☐	60	0.724828	Honda|Accord	1996	31068	$14598	🟩	Vernon,CA	90058	37	Similar Cars
☐	5068	0.709921	Honda|Accord	1997	27812	$14998	🟩	Mountain View,CA	94040	396	Similar Cars
☐	4283	0.709921	Honda|Accord	1997	57823	$14998	🟩	Costa Mesa,CA	92626	7	Similar Cars
☐	638	0.709554	Honda|Accord	1997	34537	$15598	🟩	Culver City,CA	90230	47	Similar Cars

Figure 9.5: Refined Results for Alice



Check box if you like the car	ID	Similarity Score	Make/Model	Year	Mileage	Price	Exterior Color	City,State	Zip	Distance (miles)	Click for similar cars in same vicinity
☐	2406	0.856700	Honda|Accord	1998	35690	$15598	⬜	Cerritos,CA	90623	20	Similar Cars
☐	3978	0.821586	Nissan|Altima	1998	30511	$14598	⬜	Irvine,CA	92612	0	Similar Cars
☐	4463	0.812688	Toyota|Camry	1998	37427	$18998	⬜	Tustin,CA	92680	3	Similar Cars
☐	4224	0.803750	Mazda|626	1998	25925	$12996	⬜	Newport Beach,CA	92657	7	Similar Cars
☐	4466	0.802091	Toyota|Camry	1998	29707	$19598	⬜	Santa Ana,CA	92703	8	Similar Cars
☐	1829	0.783315	Honda|Accord	1999	34408	$15598	⬜	Venice,CA	90291	51	Similar Cars
☐	1519	0.767848	Honda|Accord	1997	36859	$16798	⬜	Pacific Palisade,CA	90272	57	Similar Cars
☐	909	0.752006	Nissan|Altima	1998	38130	$12598	⬜	Downey,CA	90240	29	Similar Cars
☐	2815	0.751606	Nissan|Maxima	2000	21962	$17998	⬜	Paramount,CA	90723	29	Similar Cars
☐	2943	0.740381	Honda|Accord	1998	39409	$18598	🟫	Paramount,CA	90723	29	Similar Cars
☐	2813	0.740381	Honda|Accord	1999	38650	$18998	🟧	Paramount,CA	90723	29	Similar Cars
☐	3909	0.733741	Toyota|Camry	1998	32118	$13598	⬜	Riverside,CA	92501	36	Similar Cars
☐	2487	0.732995	Nissan|Maxima	2000	20721	$18598	⬜	Torrance,CA	90501	37	Similar Cars
☐	149	0.732906	Nissan|Altima	2001	10598	$14998	⬜	Los Angeles,CA	90001	37	Similar Cars
☐	469	0.732906	Toyota|Camry	1999	18111	$15998	⬜	Los Angeles,CA	90001	37	Similar Cars
☐	487	0.732906	Toyota|Camry	1998	33944	$14998	⬜	Los Angeles,CA	90001	37	Similar Cars
☐	386	0.732453	Nissan|Altima	1998	33702	$14598	⬜	Vernon,CA	90058	37	Similar Cars
☐	426	0.732453	Nissan|Maxima	2000	17039	$18998	⬜	Vernon,CA	90058	37	Similar Cars
☑	4227	0.732421	Acura|Integra	1999	18693	$16998	⬜	Costa Mesa,CA	92626	7	Similar Cars
☐	4376	0.731499	Infiniti|I30	1997	39238	$17998	⬜	Santa Ana,CA	92703	8	Similar Cars

Figure 9.6: First Screen for Cathy



Check box if you like the car	ID	Similarity Score	Make/Model	Year	Mileage	Price	Exterior Color	City,State	Zip	Distance (miles)	Click for similar cars in same vicinity
☑	4227	0.894315	Acura|Integra	1999	18693	$16998	⬜	Costa Mesa,CA	92626	7	Similar Cars
☐	4438	0.871034	Acura|Integra	1998	30502	$15998	⬜	Santa Ana,CA	92703	8	Similar Cars
☑	2889	0.834683	Acura|Integra	2000	8059	$17598	⬜	Paramount,CA	90723	29	Similar Cars
☑	925	0.833172	Acura|Integra	1999	18889	$17998	⬜	Downey,CA	90240	29	Similar Cars
☐	2406	0.803252	Honda|Accord	1998	35690	$15598	⬜	Cerritos,CA	90623	20	Similar Cars
☑	868	0.792645	Acura|Integra	2000	16019	$18598	⬜	El Segundo,CA	90245	45	Similar Cars
☐	2582	0.790959	Acura|Integra	1998	15592	$15998	⬛	Cerritos,CA	90623	20	Similar Cars
☐	2728	0.784618	Acura|Integra	1998	28705	$15998	🟩	Long Beach,CA	90803	24	Similar Cars
☐	1856	0.781755	Acura|Integra	1999	18770	$17598	⬜	Marina del Rey,CA	90292	50	Similar Cars
☐	1967	0.778616	Acura|Integra	1998	29890	$14998	⬜	Venice,CA	90291	51	Similar Cars
☐	671	0.777793	Acura|Integra	2000	7202	$16998	⬜	Beverly Hills,CA	90210	50	Similar Cars
☐	885	0.766801	Acura|Integra	1998	47189	$14598	⬛	Downey,CA	90240	29	Similar Cars
☐	1665	0.761404	Acura|Integra	1998	36430	$15598	⬜	Pacific Palisade,CA	90272	57	Similar Cars
☐	141	0.747445	Acura|Integra	2000	9622	$18998	⬛	Los Angeles,CA	90001	37	Similar Cars
☐	2943	0.747419	Honda|Accord	1998	39409	$18598	🟫	Paramount,CA	90723	29	Similar Cars
☐	2813	0.747419	Honda|Accord	1999	38650	$18998	🟧	Paramount,CA	90723	29	Similar Cars
☐	1565	0.744811	Acura|Integra	1998	33448	$15598	⬜	Pacific Palisade,CA	90272	57	Similar Cars
☐	1443	0.744811	Acura|Integra	1998	42698	$14598	⬜	Pacific Palisade,CA	90272	57	Similar Cars
☐	4201	0.740680	Honda|Accord	1997	55512	$13998	⬜	Costa Mesa,CA	92626	7	Similar Cars
☐	3966	0.732511	Honda|Accord	1996	56289	$13598	⬜	Irvine,CA	92612	0	Similar Cars

Figure 9.7: Refined Results for Alice

- Functionality. We provide native support for similarity search and query refinement. Similarity search can be implemented as a wrapper on top of a database, but doing this for query refinement is much more difficult.
- Eliminate guesswork. We incrementally compute the next best set of answers. Therefore we eliminate the feared “there are 632 answers to your query” phenomenon and just keep returning the next best set of answers if and when the user asks for it. On the other hand, the wrapper approach constructs a query range for each attribute and executes the range query in one go. Properly guessing the query ranges is nearly impossible since (1) different queries need different ranges (2) under-guessing eliminates good candidate answers (false negatives), and (3) over-guessing includes too many irrelevant answers (false positives), the sorting of which slows retrieval and produces the dreaded “there are 632 answers to your query” message.
- Speed. By natively supporting similarity retrieval, we can evaluate a query much faster than traditional databases. For refined queries, we reduce the response time by reusing old answers.

In the remainder of section, we compare the Informix wrapper approach and our approach for two queries.

Query 1 Consider a user in Livermore, CA that wants to buy a 1997 Honda Accord for around \$12,000. This user is however open to travel towards the Bay area where there is a higher concentration on vehicles. The search conditions of the above query are: (make=Honda) (model=Accord) (price=\$12000) (year=1997) (location=Livermore, CA, 94550).

We first run an equivalent exact query in Informix, i.e., range condition only for price (equality condition for others). This is consistent with current practices where a range of years has to be specified explicitly, and location is bounded by a fixed distance from the users zip code. This exact query returned just 4 answers as there were no cars very close to Livermore in this dataset. Next we ran the query with relaxed conditions on each attribute, i.e., range conditions for year, location, price, etc. The ranges were adjusted such that the similarity functions return 0 for values outside of these ranges. This results in a very large numbers answers that need to be scored and sorted, making it extremely slow. Then, we ran the query in our database followed by one iteration of query refinement. As a final experiment, we tried to adjust the ranges for the wrapper so that there are about 40 answers (corresponding to 4 screens of results). We had to adjust the ranges 3 times to get reasonably close.

The results for this query are shown in table 9.1. The results for Informix are generally much worse than for our system, both in terms of speed and well as the quality of answers. In terms of speed, Informix is slow as it first has to retrieve all answers that satisfy the specified ranges, and only then can it start to sort on similarity values. Informix relaxed is the slowest among all the wrapper approaches as the number of satisfying answers is the highest in this case. We, on the other hand, compute the answers incrementally (e.g., top 10 answers initially, next 10 when the user asks for the next 10, etc.) and is hence much faster. In this case, refinement is as fast as an initial query, but this includes the query reformulation time, the execution time is less than before.

Query	**Execution time**	**Number of answers**
Informix exact	4 seconds	4
Informix relaxed	160 seconds	406
Our database	0.8 seconds	40 (as requested by user)
Our database (refined)	0.8 seconds	40 (as requested by user)
Informix adjusted	30 seconds	51 (closer to 40 requested)

Table 9.1: Informix vs. Our System for *Query 1*

Query	**Execution time**	**Number of answers**
Informix exact	3 seconds	2 answers
Informix relaxed	183 seconds	447 answers
Our database	0.8 seconds	40 answers (as requested by user)
Our database (refined)	0.8 seconds	40 answers (as requested by user)
Informix adjusted	23 seconds	46 answers (closer to 40 requested)

Table 9.2: Informix vs. Our System for *Query 2*

In terms of quality, Informix exact produced the worst results as it left out many relevant vehicles. Informix adjusted produced better quality results in comparison to Informix exact but still missed some relevant vehicles. For example, a 1997 Honda Accord in Stanford, CA, priced at \$13600 (tid=5811), arguably a good answer is listed by us in 13th place, but is absent from the 51 answers of the Informix adjusted approach. In fact, the Informix adjusted approach did not retrieve 11 of the top 40 answers due to imperfect ranges; this illustrates the difficulty of selecting the right ranges for all the attributes and how easy it is to lose valuable answers in the wrapper approach. The Informix relaxed approach avoids such dismissals by fully relaxing the ranges and hence produced the same answers as our system (since they have the same similarity functions) but, as mentioned before, is extremely slow.

Query 2 Consider a user looking for a blue Honda Accord around Irvine, CA, 92612.

Repeating the previous experiment, we have the results shown in table 9.2. This example required 6 rounds of range adjustments to get within 20% of 40 top answers (as returned by us), with results ranging from 24 to 120 answers depending on the setting of ranges.

Chapter 10

Conclusions

Extensions to current systems in supporting abstract data types and their processing and optimization has received a large amount of research attention, yet another challenge facing existing systems (which has not received much research attention) is the restrictive nature of the retrieval model supported. Database systems currently support only the very basic precise retrieval model outlined in section 2.

We aim to extend database systems from traditional relational databases into extensible object-relational systems that facilitate the development of applications that require storage and retrieval of objects based on their content. Retrieval is performed based on computing similarity between the objects and queries based on their content feature values producing results that are ranked based on the computed similarity values.

A very important aspect of content based retrieval is the process of query refinement. Query refinement is important due to the subjectivity of human perception and since it may be difficult to construct a good starting query in exploratory data browsing. Refinement is a very different paradigm compared to the existing database interface that does not allow for iterative queries given its precise query paradigm. Refinement is not just a matter of adding an algorithm to learn a users intention from feedback. It changes the data model, indexing mechanism, and query processing. In a typical refinement, a subsequent query is very similar to the previous one. This has very profound impact on query processing and optimizations. Research on incorporating refinement in databases is in its infancy and we believe it is among the most important directions of future work.

We are building a prototype system that implements the content based retrieval approach we advocate and evaluate the usefulness of such a system from qualitative and quantitative points of view. The increased functionality offered by our model introduces a data model where the correctness of answers is ill-defined and thus requires a qualitative evaluation involving the users perception of and interaction with the system. Conversely, ample optimization opportunities exist in such a scenario where considerable overlap between successive queries exists. We are confident that our approach fulfills many requirements faced by current applications.

Appendix A

Image Feature Descriptions

A.1 Introduction

The retrieval performance of any similarity matching system is inherently limited by the nature and the quality of the features used to represent the objects content. Feature representations for various domains such as image retrieval, time-series retrieval, video retrieval and others are a very active research area.

Under the MARS umbrella, the goal of our work is intimately tied to image retrieval. In this appendix we give an overview of the features we extract from images and their corresponding similarity functions. We have focussed on two different modes of characterizing images: the color space, the texture space, and the shape space. For color and texture, we adapted the relevant work from the literature to our context, these are discussed in sections A.2 and A.3. For shape, we developed a new feature representation, which we present in section A.4. Note that all our image features are multidimensional feature vectors.

A.2 Color

Color is one of the most dominant features in images. The typical color encoding used in color images is in the RGB format which represents the amount of red, green, and blue colors of each pixel. However the RGB color scheme is geared towards how hardware manages color, not the human perception of color. Alternate color schemes are introduced to capture human perception; for example, HSV (hue, saturation, and value) and $L^*u^*v^*$ (lightness, chromatic coordinates). Interested readers are referred to [15] for details. We use the *color histogram* feature in the HSV space and ignore the V component as it is influenced by lighting of the scene.

Color histogram extraction discretizes image colors into bins and counts how many pixels belong to each color bin (see Figure A.1). M. Swain and D. Ballard [158] proposed *Histogram Intersection* (HI) as the similarity function for computing the similarity between two images based on their color

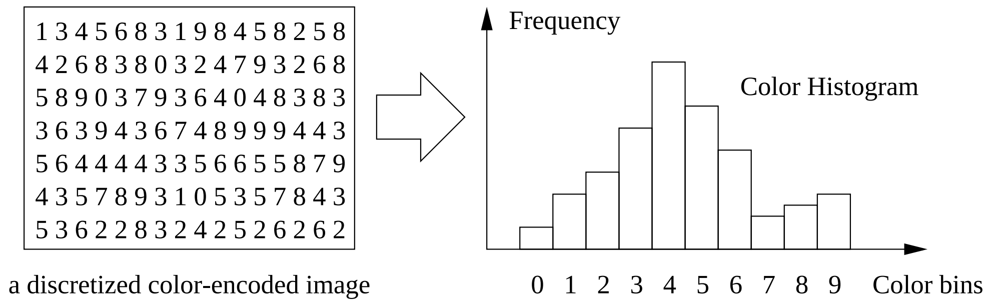


Figure A.1: Color Histogram

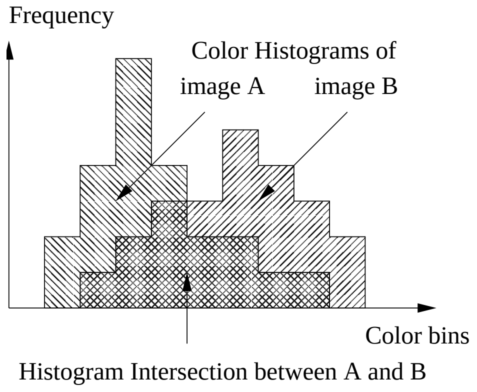


Figure A.2: Histogram Intersection

histograms. Histogram intersection is defined as follows:

$$HI(i,j) = \sum_{k=1}^{n} \min(H_i(k), H_j(k)) \tag{A.1}$$

where H_i and H_j are color histograms of images i and j, $H(k)$ is the number of pixels in bin k, and n is the total number of histogram bins (see Figure A.2). Note that the total number of pixels in each image is normalized to 1 to establish fairness among images of different sizes. A color histogram of an image can be viewed as an n-dimensional vector and the histogram intersection of two color histograms is a linear function of Manhattan distance (L_1-distance) between the two color histogram vectors as shown in (A.2).

$$HI(i,j) = 1 - \frac{L_1(i,j)}{2} \qquad \text{where} \qquad L_1(i,j) = \sum_{k=1}^{n} \mid H_i(k) - H_j(k) \mid \tag{A.2}$$

Color histogram is commonly used since it is simple and fast to compute. However, since color histogram ignores spatial information of colors, images with very different layouts may have similar color histogram representations which cause false admission in similarity retrieval.

A.3 Texture

Texture is one of the most difficult features for a person to describe in words. Texture is not a property of a single pixel in an image but rather a visual pattern formed by a contiguous region of pixels in the image. Many existing texture extraction algorithms exist in the image processing literature. Different approaches capture different properties of textures. For example, *Co-occurrence Matrix* [58, 15] captures energy, entropy, contrast, and homogeneity among other properties in an image. *Wold Decomposition* [96] captures repetitiveness, directionality, and complexity of an image. *Wavelet Texture* [99] captures the patterns of color differences of an image. We use the Co-occurrence Matrix texture representation as an example of a texture extraction algorithm.

In the Co-occurrence Matrix texture approach, the color of each pixel of an image is converted to a 16-grey-scale value. Four 16-by-16 co-occurrence matrices $C_k, k = 1, 2, ..., 4$ are constructed to capture the spatial relationship among pixels and their neighbors in each of the four directions: up-down, left-right, and the two diagonals. That is, $C_k[i, j]$ indicates the relative frequency at which two pixels of grey-scale value i and j are neighbors of each other in the k direction. Based on these matrices of co-occurrence, a subset of several characteristic properties is computed for each of the matrices:

$$energy_k = \sum_{i,j} C_k[i,j]^2 \tag{A.3}$$

$$entropy_k = \sum_{i,j} C_k[i,j] \ \log(C_k[i,j]) \tag{A.4}$$

$$contrast_k = \sum_{i,j} (i-j)^2 \ C_k[i,j] \tag{A.5}$$

$$homogeneity_k = \sum_{i,j} \frac{C_k[i,j]}{1+|i-j|} \tag{A.6}$$

These values form a 16–dimensional vector and the Euclidean distance function between these vectors is used to compute the dis-similarity between corresponding images. For details on converting distance based metrics to similarity values, see section A.5.

A.4 Shape

Large repositories of digital images are becoming increasingly common in many application areas such as e-commerce, medicine, media/entertainment, education and manufacturing. There is an increasing application need to search these repositories based on their visual content. For example, in e-commerce applications, shoppers would like to find items in the store based on, in addition to other criteria like category and price, visual criteria i.e. items that look like a selected item (e.g., all shirts having the same color/pattern as a chosen one). To address this need, we are building the

Multimedia Analysis and Retrieval System (MARS), a system for effective and efficient content-based searching and browsing of large scale multimedia repositories [115]. MARS represents the content of images using visual features like color, texture and shape along with textual descriptions. The similarity between two images is defined as a combination of their similarities based on the individual features [115].

One of the most important features that represent the visual content of an image is the *shape* of the object(s) in an image [43, 97, 101, 79]. In this paper, we address the problem of similar shape retrieval in MARS. We propose a novel adaptive resolution (AR) representation of 2-d shapes. We show that our representation is invariant to scale, translation and rotation. We show how each shape, represented by AR, can be mapped to a point in a high dimensional space and can hence be indexed using a multidimensional index structure [25]. We define a distance measure for shapes and discuss how similarity queries, based on the above distance measure, can be executed efficiently using the index structure. The experimental results demonstrate the effectiveness of our approach and its superiority to the fixed resolution (FR) technique previously proposed in the literature.

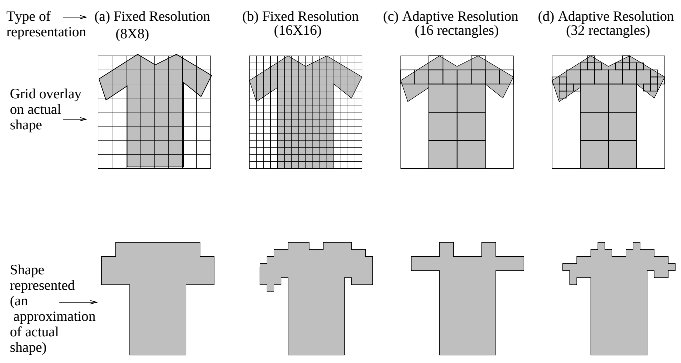
A.4.1 Related Work

In this section, we describe the fixed resolution approach and other prior work on shape retrieval.

Fixed Resolution (FR) Representation

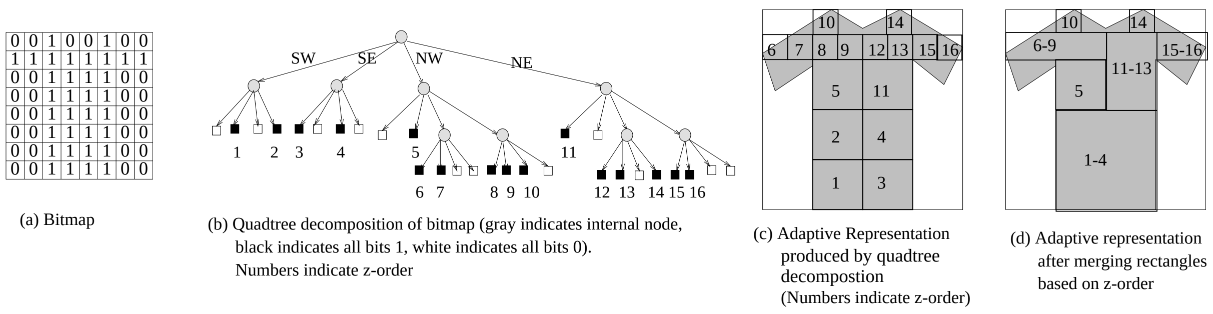
In the FR approach proposed by Lu and Sajjanhar [97], a grid, just big enough to cover the entire shape, is overlaid on the shape. Each cell of the grid has the same size (hence the name fixed resolution). For example, in figure A.3(a), the shape is overlaid with a 8×8 grid. If we assume the grid to be 256×256 pixels, each cell is 32×32 pixels in size. Some grid cells are fully or partially covered by the shape and some are not. A bitmap is derived for the shape by assigning 1 to any cell with more than 15% of the pixels covered by the shape, and 0 to each of the other cells. The shape represented by the bitmap is shown in figure A.3(a) (below the 8×8 grid overlay). The quality of the representation (i.e. how closely it approximates the actual shape) improves as we go to higher resolutions. Figure A.3(b) shows that a higher resolution bitmap (16×16 grid) represents the better (i.e. closer approximation to the original shape) than the 8×8 representation.

To support similarity queries, [97] defines a distance measure between shapes. The distance between two shapes is defined as the number of bits by which their bitmaps differ from each other. A similarity query computes the distance of the query shape from every shape in the database and returns the k closest matches as answers to the user. [97] shows that the higher the resolution, the more closely it approximates the actual shape, higher the accuracy of the answers (in terms of satisfying the information need of the user). But high resolution also raises the query cost: at higher resolutions, we need more bits to represent each shape which increases both the I/O cost (as we need to scan a larger sized database) as well as the CPU cost (as we need to compute distances between longer bit sequences) of the query. The choice of the resolution thus presents a tradeoff



Type of representation
(a) Fixed Resolution (8X8)
(b) Fixed Resolution (16X16)
(c) Adaptive Resolution (16 rectangles)
(d) Adaptive Resolution (32 rectangles)
Grid overlay on actual shape
Shape represented (an approximation of actual shape)

Figure A.3: Fixed Resolution and Adaptive Resolution Representations



0 0 1 0 0 1 0 0
1 1 1 1 1 1 1 1
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
(a) Bitmap
SW
SE
NW
NE
1 2 3 4 5 11
6 7 8 9 10 12 13 14 15 16
(b) Quadtree decomposition of bitmap (gray indicates internal node, black indicates all bits 1, white indicates all bits 0). Numbers indicate z-order
10 14
6 7 8 9 12 13 15 16
5 11
2 4
1 3
(c) Adaptive Representation produced by quadtree decompostion (Numbers indicate z-order)
10 14
6-9
15-16
11-13
5
1-4
(d) Adaptive representation after merging rectangles based on z-order

Figure A.4: Adaptive Resolution Representations

between the query cost and accuracy.

Indexing in FR Approach

There is no obvious way to index bitmaps. The only way to answer a similarity query is to sequentially scan the entire database i.e. compute the distance of the query from every item in the database. This technique is not scalable to large databases consisting of millions of shapes as it may take minutes or even hours to answer a query. To circumvent the problem, we propose to index the shapes using a multidimensional index structure (e.g., R-tree, SS-tree, Hybrid Tree). The index structure would reduce the cost of the query from linear (of sequential scan) to logarithmic to the size of the database, thus making the similarity queries scale to large sized databases. To use the index, instead of assigning a bit to a grid cell, we assign a count to each cell: the count being the number of pixels in the cell that are covered by the shape. For a 8×8 grid, we will get 64 values for each shape, thus mapping each shape to a point in a 64-dimensional space. The shapes can now be indexed using a 64-d point index structure. For a $n1 \times n2$ grid, the number of dimensions $d = n1.n2$. We can use a suitable L_p metric as the distance measure between the d-dimensional vectors. Given a query which is also a point in the d-dimensional space, we can use the index structure to quickly find the k shapes that are closest to the query using the standard k nearest neighbor algorithm

[70]. As mentioned before, higher the resolution, more the dimensionality, higher the accuracy of the answers, but higher the execution cost of the query. We will refer to this count-based (and not the bit-based) representation as the FR representation for the rest of the paper.

Other Shape Retrieval Techniques

Other shape representation techniques include Fourier descriptors [135], moment descriptors [43], boundary points [101] and rectangle decomposition [79]. Recent studies have shown that the FR approach performs better than most of these approaches [97].

A.4.2 Adaptive Resolution Approach

Rotation, Scale and Translation Normalization

Before we present the new shape representation, we describe how we normalize the shape to make it invariant to rotation, scale and translation. Our normalization strategy is similar to that of [97] developed for the FR representation. To guarantee rotation invariance, we need to convert an arbitrarily oriented shape into a unique common orientation. We first find the major axis of the shape i.e. the straight line segment joining the two points P_1 and P_2 on the boundary farthest away from each other [97]. Then we rotate the shape so that its major axis is parallel to the x-axis. This orientation is still not unique as there are two possibilities: P_1 can be on the left or on the right. We solve the problem by computing the centroid of the polygon and making sure that the centroid is below the major axis, thus guaranteeing an unique orientation. Let us now consider scale invariance. We define the bounding rectangle (BR) of a shape as the rectangle with sides parallel to the x and y axes just large enough to cover the entire shape (after rotation). Note that the width of the BR is equal to the length of the major axis. To achieve scale invariance, we proportionally scale all shapes so that their BRs have the same fixed width. In this paper, we fix the width of the BR to 256 pixels. We make the BR a square by fixing its height to 256 pixels as well; non-square shapes are handled by placing the shape at the bottom of the BR and padding zeroes in the remaining (upper) part of the BR. The shape is translation invariant as it is represented with respect to its BR (i.e. lower left corner of BR is considered (0,0)).

Adaptive Resolution Representation

The problem with the FR representation is that it uses the same resolution to represent the entire shape. There are certain portions of the shape where low resolution is sufficient i.e. increasing the resolution does not improve the quality of the representation for these portions of the shape (e.g., the lower rectangular portion of the shape in figures A.3(a) and (b)). Using high resolution for these regions increases the number of dimensions without improving the query accuracy. On the other hand, there are portions of the shape (e.g., the upper portion of the shape in figures A.3(a) and (b)) where higher resolution improves the quality of the representation significantly. For these regions, the improvement in query accuracy obtained by using high resolution is worth the extra cost of having more dimensions. Also, high resolution is usually not necessary for the interior of a shape but is important near the border of the shape. Having the same resolution for the entire shape is wasteful in terms of the number of dimensions used to represent the shape and hence the query cost.

To overcome the shortcomings of the FR representation, we propose an *adaptive resolution* (AR) representation of shapes i.e. a representation where the resolution of the grid cells varies from one portion of the shape to another, having higher resolution where it improves the quality of representation and lower resolution where it does not. An adaptive representation of the same shape is shown in figure A.3(c). It uses 16 grid cells to represent the entire shape but the cells have different resolutions (i.e. sizes). The cells in the lower portion and interior of the shape have lower resolution (i.e. larger size) while those in the upper portion and near the borders have higher resolution (i.e. smaller size). Figure A.3(d) shows another adaptive representation of the same shape with 32 grid cells. Note that as the number of cells increases, more cells are added to the portion of the shape where higher resolution is required while the other portions remain unchanged.

Computing AR Representation Using Quadtree Decomposition

We compute the AR representation of a shape by applying quadtree decomposition on the bitmap representation of the shape. The bitmap is constructed in the same way as the FR approach discussed in [97] (cf. section A.4.1). We use the highest resolution bitmap for the decomposition (i.e. each grid cell is 1×1 pixels) but lower resolution bitmaps could be used as well. The decomposition is based on successive subdivision of the bitmap into four equal-size quadrants. If a bitmap-quadrant does not consist entirely of 1s or entirely of 0s (i.e. the shape "partially covers" the quadrant), it is recursively subdivided into smaller and smaller quadrants until we reach bitmap-quadrants, possibly 1×1 pixels in size, that consist entirely of 1s or entirely of 0s (termination condition of the recursion). Figure A.4(a) shows a 8×8 bitmap of the shape in figure A.3 and figure A.4(b) shows the quadtree decomposition of the bitmap. Each node in the quadtree covers a rectangular (always square) region of the bitmap. The level of the node in the quad tree determines the size of the rectangle. The internal nodes (shown by gray circles) represent "partially covered" regions, the leaf nodes shown by white boxes represent regions with all 0s while the leaf nodes shown by black boxes represent regions with all 1s. The "all 1" regions are used to represent the shape. Figure A.4(b) has 16 such rectangular regions and the shape represented is shown in figure A.4(c). Since we perform the quadtree decomposition on the 256×256 bitmap, the number of black leaf nodes in the quadtree is usually far more than the number n of rectangles we want to choose to represent the shape. In that case, we choose the n largest rectangles i.e. we do not choose any black leaf node at level i unless we have chosen all the black leaf nodes at level $j, j < i$ where the levels are numbered in increasing order from top to bottom. In this way, we can cover the bulk of the shape with a few rectangles (i.e. small values of n) and add details to the shape as we add more rectangles (i.e. larger values of n).

```
Distance(int *shape1, int *shape2)

i=0, j=0;
while ($i < n$ AND $j < n$)
      if $i$th rect r1 of shape1 overlaps with $j$th rect r2 of shape2
         common_area += Overlap_Area(r1, r2);
         if (r1 is bigger than r2) j++;
         else i++;
      else // r1 and r2 do not overlap
         if (ZValue(r1) > ZValue(r2)) j++;
         else i++;
area_of_shape1 = $\Sigma_{i=0}^{(n-1)}$ Area of $i$th rectangle;
area_of_shape2 = $\Sigma_{i=0}^{(n-1)}$ Area of $i$th rectangle;
union_area = area_of_shape1 + area_of_shape2 - common_area;
distance = 1 - $\frac{common\_area}{union\_area}$;
return distance;
```

Table A.1: Computing Distance between two Shapes

Indexing

After the n rectangles are chosen, we sort them based on z-order. The number sequence assigned to the black leaf nodes of the quad tree in figure A.4(b) represent the z-order. The same numbers are shown on the corresponding rectangles in figure A.4(c). Note that the z-ordering is simply a left-to-right ordering of the n selected black leaf nodes in the quad tree. We represent the shape as a *sequence* of the n rectangles. Since the rectangles are always squares, we can describe each rectangles by 3 numbers: its center $C = (C_x, C_y)$ and its size (i.e. side length) S. We represent the shape as a sequence of $3n$ numbers where $3i$, $3i+1$ and $3i+2$ numbers represent the C_x, C_y and S of the ith rectangle $(0 \leq i \leq (n-1))$ in the n-sequence. We have thus mapped each shape to a point in $3n$-dimensional space. We can now index the shapes using a $3n$-dimensional index structure. The choice of n depends on the desired dimensionality d of the index structure i.e. $n = \frac{d}{3}$.

Executing Similarity Queries Using Multidimensional Index Structure

To support similarity queries, we must first choose a distance measure between shapes. We choose the popular "area difference" distance measure previously used in [97, 79]. The area difference is the area of the regions where the two shapes do not match when they are overlaid on each other. To normalize the measure, we divide the area difference by the area covered by the two shapes together i.e. area of the union of the two shapes. To be able to answer similarity queries using a multidimensional index structure, we should be able to efficiently compute (1) the distance between two points i.e. between two shapes represented using the AR representation and (2) the minimum distance (MINDIST) between a point and a node of the multidimensional index structure

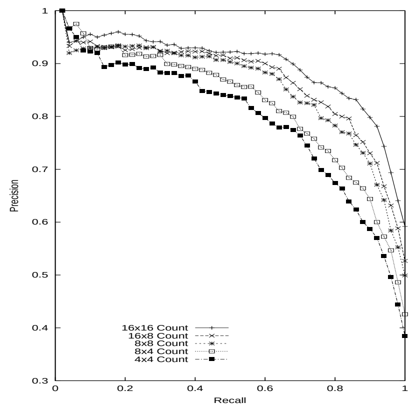


Figure A.5: Precision-recall Graph for FR Technique

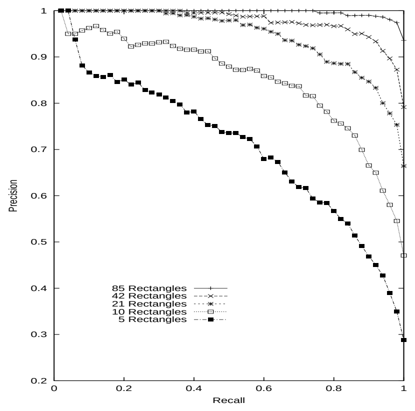


Figure A.6: Precision-recall Graph for AR Technique

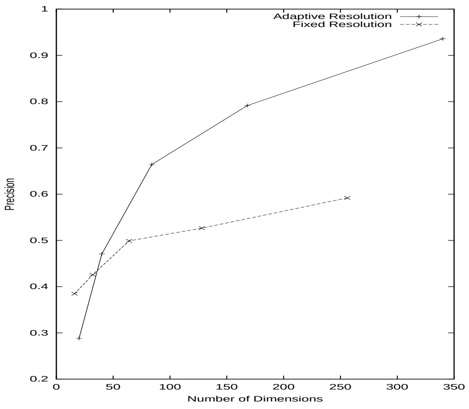


Figure A.7: Comparison of FR and AR Techniques at 100% Recall

[25]. Once we can compute the above distances, we can answer a similarity query by executing the k-NN algorithm on the multidimensional index structure [70]. The algorithm works as follows. It maintains the nodes and objects of the index structure in the priority queue in increasing order of their distances from the query and uses the queue to traverse the tree in the same order. At each step, it pops the item from the top of the queue: if it is an object, it is added to the result list, if it is a node, it computes, using the above distance functions, the distance of each of its children from the query and pushes it back to the queue. The algorithm stops when the result set contains k objects. We first present the function to compute the distance between two points. Given two shapes $s1$ and $s2$ consisting of n rectangles each (i.e. represented as $3n$-dimensional points), a naive way to compute the distance is to compare all pairs of rectangles and compute the distance between them. This approach is computationally expensive ($O(n^2)$). We exploit the following properties of Z-ordering to compute the distance in $O(n)$ time. Let $r1$ and $r2$ be two rectangles of $s1$ and $s2$ respectively. First, if $r1$ and $r2$ do not overlap with each other and $ZValue(r1) > ZValue(r2)$ and $r1'$ is a rectangle in $s1$ that appears after $r1$ (i.e. $ZValue(r1') > ZValue(r1)$), then $r1'$ and $r2$ do not overlap with each other. Second, if $r1$ and $r2$ overlap with each other and $r1$ is larger than $r2$ (i.e. $r1$ is totally covers $r2$) and $r1'$ is a rectangle in $s1$ that appears after $r1$ (i.e. $ZValue(r1') > ZValue(r1)$), then $r1'$ and $r2$ do not overlap with each other. The procedure to compute the distance is shown in Table A.1. We have also designed an algorithm to compute the MINDIST i.e. the minimum distance between a point and node of the index structure. We can now answer similarity queries efficiently by executing it as a k-NN query on the multidimensional index structure.

Optimization

The quality of the representation (and hence the accuracy of the answers) increases with the number of rectangles, but so does the dimensionality and hence the query cost. We present an optimization that increases the number of rectangles without increasing the number of dimensions. We merge rectangles together if they are (1) "mergeable" i.e. produce a rectangle when merged and (2) appear consecutively in the z-ordered sequence. Figure A.4(d) shows the set of rectangles in figure A.4(c) after the merging. Since this representation is more compact (i.e. we can represent the same shape with less number of rectangles), for a given choice of dimensionality d, we can represent the shape more accurately. The merging based on z-order ensures that the distance functions described above can be used with some minor modifications. Unlike in the unmerged representation, the rectangles in the merged representation can be non-squares; hence, we need to store two sizes S_x and S_y for each rectangle instead of one. To represent n rectangles, we need $4n$ numbers instead of $3n$. For a desired dimensionality d, the number of rectangles to choose to represent the shape is $\frac{d}{4}$ instead of $\frac{d}{3}$.

A.4.3 Experiments

We conducted several experiments to evaluate the effectiveness of the AR representation and compare it with the FR representation. For our experiments, we used the "islands" file in the polygon dataset of the Sequoia benchmark.[1] The dataset contains 21,021 shapes. For a given query, we generate the ground truth by executing the query against the highest resolution bitmap representation of the shape (i.e. each grid cell is 1×1 pixel) and retrieve the top k answers. We refer to these answers as the *relevant set*. We then execute the query against a given low resolution FR representation using the index structure and retrieve the top k answers. We refer to these answers as the *retrieved set*. We compare the relevant and retrieved sets for various values of k (we vary k from 10 to 100) and we plot the precision (defined as $\frac{|relevant \cap retrieved|}{|retrieved|}$) and recall (defined as $\frac{|relevant \cap retrieved|}{|relevant|}$) graphs for various resolutions (4×4, 8×4, 8×8, 16×8 and 16×16). The result is shown in figure A.5. All the measurements are averaged over 100 queries. The graph shows that the quality of the answers improves as the resolution increases. We repeat the above experiment for the AR representation. The result is shown in figure A.6. The graph shows that the quality of the answers improves with the increase in the number of rectangles. Note that we are doubling the number of dimensions at each step in both cases but the improvement in the quality of answers at each step is more significant in the AR technique compared to the FR technique. The reason is that in the AR case, the additional rectangles concentrate on improving the representation *only* where it is necessary. On the other hand, in the FR case, the resolution is increased equally all over, as much in the unnecessary portions as in the necessary ones, thus diluting the effect and not improving the quality of representation as much as in AR case.

We compare the two techniques in terms of the quality of retrieval when the same number of dimensions are used to represent the shape. For a given recall, we compute the precision of the two techniques at a given dimensionality. Note that for the FR presentation of $n1 \times n2$ resolution, the dimensionality is $n1.n2$ while for the AR approach with n rectangles, the dimensionality is $4n$.

[1] Available online at http://s2k-ftp.cs.berkeley.edu:8000/sequoia/benchmark/polygon/.

In figure A.7, we plot the precision at 100% recall for various dimensionalities for both techniques. The AR technique significantly outperforms the FR technique in terms of precision at almost all dimensionalities. For example, at 100 dimensions, the AR technique has about 70% precision while the FR technique has about 50% precision. We observed similar behaviour at other values of recall. This shows that AR is a more compact representation i.e. with the same number of dimensions, AR approximates the original shape better than FR and hence provides higher precision. Assuming that the query cost is proportional to the number of dimensions used[2], the AR technique provides significantly better quality answers at the same cost and is hence a better approach to shape retrieval.

A.4.4 Conclusion

Similar shape retrieval is an important problem with a wide range of applications. In this paper, we have presented a novel adaptive resolution approach to representing 2-d shapes. The representation is invariant to scale, translation and rotation. With the proposed representation, we can index the shapes using a multidimensional index structure and can thus support efficient similarity retrieval. Our experiments show that, for the same query cost, the adaptive technique provides significantly better quality answers compared to the fixed resolution representation and is hence a better approach to shape retrieval.

A.5 Feature Sequence Normalization

Depending on the extracted feature, some normalization may be needed. The normalization process serves two purposes:

1. It puts an equal emphasis on each feature element within a feature vector. To see the importance of this, notice that in the texture representation, the feature elements may be different physical quantities. Their magnitudes can vary drastically, thereby biasing the Euclidean distance measure. This is overcome by the process of *intra-feature* normalization.

2. It maps the distance values of the query from each atomic feature into the range [0,1] so that they can be interpreted as the degree of membership in the fuzzy model or relevance probability in the probability model. While some similarity functions naturally return a value in the range of [0, 1], e.g. the color histogram intersection; others do not, e.g. the Euclidean distance used in texture. In the latter case the distances need to be converted to the range of [0, 1] before they can be used. This is referred to as *inter-feature normalization.*

[2]We have performed experiments using a multidimensional index structure and measured the actual I/O and CPU cost for the two techniques. Our experiments validate the claim that the query cost is indeed proportional (actually super-linearly) to the number of dimensions used.

A.5.1 Intra-feature Normalization

This normalization process is only needed for features using a *vector based* representation, as in the case of the wavelet texture feature representation.

For the vector based feature representation, let $F = [f_1, f_2, ..., f_j, ..., f_N]$ be the feature vector, where N is the number of feature elements in the feature vector and $I_1, I_2, \ldots, I_M$ be the images. For image I_i, we refer to the corresponding feature F as $F_i = [f_{i,1}, f_{i,2}, ..., f_{i,j}, ..., f_{i,N}]$. Since there are M images in the database, we can form a $M \times N$ feature matrix $F = f_{i,j}$, where $f_{i,j}$ is the jth feature element in feature vector F_i. Each column of F is a length-M sequence of the jth feature element, represented as F_j. The goal is to normalize the entries in each column to the same range so as to ensure that each individual feature element receives equal weight in determining the Euclidean distance between the two vectors.

Assuming the feature sequence F_j to be a Gaussian sequence, we compute the mean m_j and standard deviation σ_j of the sequence. We then normalize the original sequence to a N(0,1) sequence as follows:

$$f'_{i,j} = \frac{f_{i,j} - m_j}{\sigma_j} \tag{A.7}$$

Note that after the Gaussian normalization, the probability of a feature element value being in the range of [-1, 1] is 68%. If we use $3\sigma_j$ in the denominator, the probability of a feature element value being in the range of [-1, 1] is approximately 99%. In practice, we can consider all of the feature element values within the range of [-1,1] by mapping the out-of-range values to either -1 or 1.

A.5.2 Inter-feature Normalization

Intra-feature normalization ensures that equal emphasis is put on each feature element within a feature vector. On the other hand, inter-feature normalization ensures equal emphasis of each feature within a composite query. The aim is to convert similarity values (or distance in some cases like wavelet) into the range [0,1].

The feature representations used in MARS are of various forms, such as vector based (wavelet texture representation), histogram based (histogram color representation), irregular (MFD shape representation), etc. The distance computations of some of these features (e.g. color histogram) naturally yield a similarity value between 0 and 1 and hence do not need additional normalization. Distance calculations in other features are normalized to produce values in the range [0,1] with the process described below.

1. For any pair of images I_i and I_j, compute the distance $D_{(i,j)}$ between them:

$$D_{(i,j)} = dist(F_{I_i}, F_{I_j}) \tag{A.8}$$

$$\begin{aligned} i, j &= 1, ..., M, \\ i &\neq j \end{aligned}$$

where F_{I_i} and F_{I_j} are the feature representations of images I_i and I_j.

2. For the $C_2^M = \frac{M \times (M-1)}{2}$ possible distance values between any pair of images, treat them as a value sequence and find the mean m and standard deviation σ of the sequence. Store m and σ in the database to be used in later normalization.

3. After a query Q is presented, compute the raw (un-normalized) distance value between Q and the images in the database. Let $s_1, ..., s_M$ denote the raw distance values.

4. Normalize the raw distance values as follows:

$$s_i' = \frac{s_i - m}{3\sigma} \tag{A.9}$$

As explained in the intra-feature normalization section, this Gaussian normalization will ensure 99% of s_i' to be within the range of [-1,1]. An additional shift will guarantee that 99% of distance values are within [0,1]:

$$s_i'' = \frac{s_i' + 1}{2} \tag{A.10}$$

After this shift, in practice, we can consider all the values within the range of [0,1], since an image whose distance from the query is greater than 1 is very dissimilar and can be considered to be at a distance of 1 without affecting retrieval.

5. Convert from distance values into similarity values. This can be accomplished by the following operation:

$$similarity_i = 1 - s_i'' \tag{A.11}$$

At the end of this normalization, all similarity values for all features have been normalized to the same range [0,1] with the following interpretation: 1 means full similarity (exact match) and 0 denotes maximum dis-similarity.

Appendix B

Modified Normalized Recall Metric

In our model, the result of a query at each step of the feedback process is a ranked list of tuples based on a similarity value that indicates how good each tuple matches the query. The metric used is designed to compare such ranked lists returned at different iterations of the feedback process.

To measure the effectiveness of the query refinement process, we run an initial query with tuples chosen at random from the collection of possible tuples for a query (named a collection) and save the ranked result as the ground truth as if it were a desired result a user would expect. This is the *Relevant* list RL_1. This process avoids the user's subjectivity and the effect of how well features capture tuple contents in the evaluation of the feedback process. The query results are recorded in the *Retrieved* list RL_2. The two lists are compared and the similarity between the two lists indicates how good the retrieval process is.

The system then performs refinement iteratively and at each iteration computes a new RL_2 query result ranked list. The modified recall values are computed based on the formula in (B.1) to compare the results from various iterations. The modified recall is defined as follows:

$$Recall_{modified}(RL_1, RL_2) = 1 - \frac{4 \sum\limits_{i \in RL_1} [rank(RL_2, i) - rank(RL_1, i)] \times \left[1 - \frac{rank(RL_1, i)}{|RL_1| + 1}\right]}{(2\widehat{N} - |RL_1| + 1) \times |RL_1|} \tag{B.1}$$

The concept behind this modified recall measure is motivated by the pitfall of the normalized recall measure in (B.2) which was described in [137] as a metric to compare two ranked lists where $rank(List, i)$ is the rank of item i in $List$.

$$Recall_{normalized}(RL_1, RL_2) = 1 - \frac{\sum\limits_{i \in RL_1} [rank(RL_2, i) - rank(RL_1, i)]}{(N - |RL_1|) \times |RL_1|} \tag{B.2}$$

The normalized recall metric is designed to compare two ranked lists, one of relevant items (the ground truth) and one of retrieved items (the result of a query). It computes the rank difference of the two lists and normalizes the result by the highest possible rank difference, thus producing a value in the range 0 (worst) to 1 (best). The metric is sensitive to the position of objects in the ranked list. This sensitivity to rank position is suitable for measuring the effectiveness of the query refinement process by comparing the relevant list to the result across feedback iterations. As the lists converge, the metric results in a better value.

The normalized recall metric is however meant to compare a list of relevant items to the fully ranked list of answers of the whole collection. The problems with the normalized recall metric are:

1. A few poorly ranked, but relevant, items have a great impact on the metric because a poorly ranked item introduces a large value of rank difference. In practice, the user views only the top $\widehat{N}$ answers with $\widehat{N} \ll N$. The impact of rank differences beyond $\widehat{N}$ is not noticeable by the user since he/she does not explore items beyond $\widehat{N}$.

2. The collection size also has a great impact on the metric. In a large collection, only a small part of it is actually explored by the user at each instance in time. The rest of the collection is of no use. Therefore, using the whole collection as an important part of the denominator in the equation obscures the difference of two good retrieved lists because the large denominator dilutes the rank difference significantly.

3. Each ranked item is equally important. That is, the best relevant item is as important as the worst relevant, but still relevant, item.

For these reasons, the normalized recall metric is modified in two ways.

- To deal with the first two problems, the retrieved set is limited to $\widehat{N}$ elements instead of the whole collection N. Items that are not found in the retrieved set are considered to have a rank of $\widehat{N}+1$: just as if they were the next item to be retrieved. In this way, a large collection does not impact negatively on the metric and the rank difference of each item is limited to the number of items that the user views/retrieves.

- To address the third problem, a weight is introduced so that rank differences of answers ranked high in the relevant set have a higher impact than lower ranked answers. A linear function is used to assign the highest weight to the rank difference of the most relevant item and the lowest weight to the rank difference of the least relevant item.

There are two ranked lists: RL_1 containing the desired ranking of the relevant objects and RL_2 containing the ranked result of a query. The modified metric in (B.1) is used. The derivation of (B.1) is from the original normalized recall which is defined as:

$$Recall_{normalized}(RL_1, RL_2) = 1 - \frac{\sum\limits_{i \in RL_1} [rank(RL_2, i) - rank(RL_1, i)]}{(N - |RL_1|) \times |RL_1|} \tag{B.3}$$

where the denominator is the maximum possible rank difference (RD_{max}) between the two ranked lists (that is, relevant list RL_1 and the retrieved list RL_2) being compared. The modified normalized recall used as a metric to compare two ranked lists makes two changes to the original normalized recall metric as discussed earlier. First, unlike normalized recall, since we do not generate a ranking for the whole collection, but rather only of the best $\widehat{N}$ answers, if a relevant object does not appear in the retrieved set, its ranking is considered to be $\widehat{N}+1$. This modifies the maximum possible rank difference between two lists as follows:

$$\begin{aligned} RD_{max} &= \widehat{N} + (\widehat{N}-1) + ... + (\widehat{N} - |RL_1| + 1) \\ &= \frac{1}{2} \times (2\widehat{N} - |RL_1| + 1) \times |RL_1| \end{aligned} \tag{B.4}$$

Next, in our metric we re-weigh the importance of rank by giving a high weight to a highly relevant item and a low weight to a lowly relevant item. For this purpose, a linear weight assignment function is used. The weight assignment function assigns a weight of 1 to the lowest relevant rank and weight of $|RL_1|$ to the highest relevant rank and is as follows:

$$w(i) = |RL_1| - rank(RL_1, i) + 1 \tag{B.5}$$

In order to keep the average of all the weights to 1 (so as to preserve the property of the original unweighted rank which can be viewed as assigning a weight of 1 to each rank), the weight is further normalized as follows:

$$w(i) = 2 \times \left(1 - \frac{rank(RL_1, i)}{|RL_1| + 1}\right) \tag{B.6}$$

We multiply the weight from Equation (B.6) with the rank difference in the original normalized recall and replace RD_{max} in (B.4). The modified normalized recall therefore becomes:

$$Recall_{modified} = 1 - \frac{4 \sum_{i \in RL_1} [rank(RL_2, i) - rank(RL_1, i)] \times \left[1 - \frac{rank(RL_1, i)}{|RL_1| + 1}\right]}{(2\widehat{N} - |RL_1| + 1) \times |RL_1|}$$

References

[1] S. Adali, P. Bonatti, M. L. Sapino, and V. S. Subrahmanian. A multi-similarity algebra. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 402–413, 1998.

[2] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In *ACM SIGMOD*, 2000.

[3] K. Alsabti, S. Ranka, and V. Singh. An efficient parallel algorithm for high dimensional similarity join. In *Proc. Int. Parallel and Distributed Processing Symp.*, Orlando, Florida, March 1998.

[4] Walid G. Aref and Hanan Samet. Optimization for spatial query processing. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proc. 17th Int. Conf. on Very Large Data Bases VLDB '92*, pages 81–90, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[5] Walid G. Aref and Hanan Samet. Cascaded spatial join algorithms with spatially sorted output. In *Proc. 4th ACM Workshop on Advances in Geographic Information Systems*, pages 17–24, 1997.

[6] Sunil Arya, David Mount, Nathan Netanyahu, Ruth Silverman, and Angela Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6), November 1998.

[7] Jeffrey R. Bach, Charles Fuller, Amarnath Gupta, Arun Hampapur, Bradley Horowitz, Rich Humphrey, Ramesh Jain, and Chiao fe Shu. The Virage Image Search Engine: An open framework for image management. In *SPIE Storage and Retrieval for Still Image and Video Databases IV.*

[8] Ricardo Baeza-Yates and Ribeiro-Neto. *Modern Information Retrieval.* ACM Press Series/Addison Wesley, New York, May 1999.

[9] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System: The Story of O2.* The Morgan Kaufmann Series in Data Management Systems, May 1992.

[10] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5), 1992.

[11] Ilaria Bartolini, Paolo Ciaccia, and Florian Waas. Feedback bypass: A new approach to interactive similarity query processing. In *27th Very Large Databases (VLDB)*, Rome, Italy, September 2001.

[12] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, May 1990.

[13] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Proc. of ICDT*, 1998.

[14] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is "Nearest Neighor" Meaningful? *Sumitted for publication*, 1998.

[15] Roberto Del Bimbo. *Visual Information Retrieval.* Morgan Kaufmann, 1999.

[16] Eric A. Brewer. When everything is searchable. *Communications of the ACM*, 44(3), March 2001.

[17] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-Trees. In Peter Buneman and Sushil Jajodia, editors, *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, Washington, DC, USA, May 26-28 1993. ACM Press.

[18] Gerth Stolting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *Lecture Notes in Computer Science, Springer-Verlag, Berlin*, volume 1432, 1998.

[19] Chris Buckley and Gerard Salton. Optimization of relevance feedback weights. In *Proc. of SIGIR'95*, 1995.

[20] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems Applications*, Valencia, Spain, 1992.

[21] M. Carey and D. Kossmann. On saying "enough already" in sql. *Proc. of SIGMOD*, 1997.

[22] Michael J. Carey and David J. DeWitt. Of objects and databases: A decate of turmoil. In *Proc. 22nd Int. Conf. on Very Large Data Bases VLDB '96*, pages 3–14, 1996.

[23] Michal J. Carey, David J. DeWitt, Daniel Frank, M. Muralikrishna, Goetz Graefe, Joel E. Richardson, and Eugene J. Shekita. The Architecture of the EXODUS extensible DBMS. In *Proc. of the 1986 Int. Workshop on Object-Oriented database systems*, pages 52–65, 1986.

[24] Chad Carson, Serge Belongie, Hayit Greenspan, and Jitendra Malik. Region-based image querying. In *Proc of IEEE Workshop on Content-based Access of Image and Video Libraries, in conjunction with IEEE CVPR '97*, 1997.

[25] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. *Proceedings of ICDE Conference*, March 1999.

[26] Kaushik Chakrabarti, Michael Ortega-Binderberger, Kriengkrai Porkaew, and Sharad Mehrotra. Similar shape retrieval in mars. In *In Proceeding of IEEE International Conference on Multimedia and Expo*, New York City, NY, USA, July 2000.

[27] Kaushik Chakrabarti, Kriengkrai Porkaew, and Sharad Mehrotra. Efficient query refinement in multimedia databases. In *IEEE ICDE*, 2000.

[28] Donald D. Chamberlin. *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann, July 1998.

[29] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In *ACM Conference on Management of Data (SIGMOD)*, June 2002.

[30] Tianhorng Chang and C.-C. Jay Kuo. Texture Analysis and Classification with Tree-Structured Wavelet Transform. *IEEE Trans. Image Proc.*, 2(4):429–441, October 1993.

[31] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

[32] Surajit Chaudhari and Luis Gravano. Optimizing Queries over Multimedia Repositories. *Proc. of SIGMOD*, 1996.

[33] Surajit Chaudhari and Luis Gravano. Evaluating top-k selection queries. In *Proc. 25th Int. Conf. on Very Large Data Bases VLDB '99*, pages 397–410, Edinburgh, Scotland, 1999.

[34] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. CoBase: A Scalable and Extensible Cooperative Information System. *Journal of Intelligent Information Systems*, 6, 1996.

[35] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proc. of VLDB*, 1997.

[36] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[37] Ingemar J Cox, MAtthew L. Miller, Thomas P. Minka, and Peter N. Yianilos. An optimized interaction strategy for bayesian relevance feedback. In *IEEE Conf. on Comp. Vis. and Pattern Recognition, Santa Barbara, California*, pages 553–558., 1998.

[38] Jitender S. Deogun and Vijay V. Raghavan. Integration of information retrieval and database management systems. *Information Processing and Management*, 24(3):303–313, 1988.

[39] Stefan Dessloch and Nelson Mattos. Integrating SQL databases with content–specific search engines. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 528–537, Athens, Greece, 1997. IBM Database Technology Institute, Santa Teresa Lab.

[40] D. Dey and S. Sarkar. A probabilistic relational model and algebra. *ACM Transactions on Database Systems*, 21(3), 1996.

[41] D. Dubois and H. Prade. A review of fuzzy set aggregation connectives. *Information Sciences*, 36:85–121, 1985.

[42] W. F. Cody et. al. Querying Multimedia Data from Multimedia Repositories by Content: the Garlic Project. *Proc. of Visual Database Systems (VDB-3)*, 1995.

[43] W. Niblack et. al. The QBIC project: Querying images by content using color, texture and shape. In *IBM Research Report*, February 1993.

[44] G. Evangelidis, D. Lomet, and B. Salzberg. The hb^{π}-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In *Proceedings of VLDB*, 1995.

[45] R. Fagin. Fuzzy queries in multimedia database systems. *Proceedings of PODS*, 1998.

[46] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *Proc. of the 15th ACM Symp. on PODS*, 1996.

[47] Ronald Fagin and Edward L. Wimmers. Incorporating User Preferences in Multimedia Queries. *International Conference on Database Theory*, 1997.

[48] C. Faloutsos, M. Flickner, W. Niblack, D. Petkovic, W. Equitz, and R. Barber. Efficient and effective querying by image content. Technical report, IBM Research Report, 1993.

[49] C. Faloutsos, M. Flocker, W. Niblack, D. Petkovic, W. Equitz, and R. Barber. Efficient and Effective Querying By Image Content. Technical Report RJ 9453 (83074), IBM Research Report, Aug. 1993.

[50] Christos Faloutsos and Douglas Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, Dept. of Computer Science, Univ. of Maryland, 1995.

[51] S. Flank, P. Martin, A. Balogh, and J. Rothey. Photofile: A Digital Library for Image Retrieval. In *Proc. 2nd Int. Conf. on Multimedia Computing and Systems*, pages 292–295, 1995.

[52] M. Flickner, Harpreet Sawhney, Wayne Niblack, and Jonathan Ashley. Query by Image and Video Content: The QBIC System. *IEEE Computer*, 28(9):23–32, September 1995.

[53] You-Chin (Gene) Fuh, Stefan Dessloch, Weidong Chen, Nelson Mattos, Brian Tran, Bruce Lindsay, Linda DeMichiel, Serge Rielau, and Danko Mannhaupt. Implementation of SQL3 Sturctured Type with Inheritance and Value Substitutability. In *Proc. 25th Int. Conf. on Very Large Data Bases VLDB '99*, pages 565–574, Edinburgh, Scotland, 1999.

[54] Norbert Fuhr. Logical and conceptual models for the integration of information retrieval and database systems. 1996.

[55] Norbert Fuhr. Models for integrated information retrieval and database systems. *Bulletin of IEEE Computer Society Technical Committee on Data Engineering*, 1996.

[56] Norbert Fuhr, Ray R. Larson, Peter Schauble, Joachim W. Schmidt, and Ulrich Thiel. Integration of information retrieval and database systems. In *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Panel Sessions, page 360, 1994.

[57] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 1999.

[58] Calvin C. Gotlieb and Herbert E. Kreyszig. Texture descriptors based on co-occurrence matrices. *Computer Vision, Graphics, and Image Processing*, 51, 1990.

[59] Goetz Graefe. Encapsulation of Parallelism in the Volcano query processing system. *Proc. of ACM SIGMOD*, 1990.

[60] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys, Vol. 25, No. 2*, 1993.

[61] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[62] J. Gray and P. Shenoy. Rules of thumb in data engineering. *http://www. research. microsoft. com/˜ gray/*, 1999.

[63] M. H. Gross, R. Koch, L. Lippert, and A. Dreger. Multiscale Image Texture Analysis in Wavelet Spaces. In *Proc. IEEE Int. Conf. on Image Proc.*, 1994.

[64] Junzhong Gu, Ulrich Thiel, and Jian Zhao. Efficient retrieval of complex objects: query processing in a hybrid db and ir system. *Proc der 1. Tagung "Information Retrieval '93", Konstanz, FRG*, pages 67–81, 1993.

[65] A. Guttman. R-tree: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[66] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf., pp. 47–57.*, 1984.

[67] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 377–388, 1989.

[68] Robert M. Haralick, K. Shanmugam, and Its'hak Dinstein. Texture Features for Image Classification. *IEEE Trans. on Sys, Man, and Cyb*, SMC-3(6), 1973.

[69] G. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 237–248, Seattle, WA, USA, June 1998.

[70] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Proceedings of SSD*, 1995.

[71] Gisly R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Advances in Spatial Databases - 4th Symposium, SSD'95, M. J. Egenhofer and J. R. Herring, Eds., Lecture Notes in Computer Science 951, Springer-Verlag, Berlin*, pages 83–95, 1995.

[72] Gisli R. Hjaltson and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. ACM SIGMOD*, pages 237–248, 1998.

[73] Martijn J. Hoogeveen and Kees van der Meer. Integration of information retrieval and database management in support of multimedia police work. *Journal of Information Science*, 20(2):79–87, 1994.

[74] M.J. Hoogeveen and K. Van der Meer. Integration of information retrieval and database management in support of multimedia police work. *Journal of Information Science*, 20(2):79–87, 1994.

[75] Thomas S. Huang, Sharad Mehrotra, and Kannan Ramchandran. Multimedia Analysis and Retrieval System (MARS). In *33rd Annual Clinic on Library Application of Data Processing "Digital Image Access and Retrieval"*, March 1996.

[76] David Hull. Improving text retrieval for the routing problem using latent semantic indexing. In *ACM Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 282–291, 1994.

[77] Informix. *Getting Started with Informix Universal Server, Version 9.1.* Informix, March 1997.

[78] Yoshiharu Ishikawa, Ravishankar Subramanya, and Christos Faloutsos. Mindreader: Querying databases through multiple examples. In *Proc. 24th Int. Conf. on Very Large Data Bases VLDB '98*, pages 218–227, 1998.

[79] H. Jagadish. A retrieval technique for similar shapes. *Proceedings of SIGMOD*, 1991.

[80] Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing and Management*, 36(2), 2000.

[81] H. Jiang, D. Montesi, and A. Elmagarmid. Videotext database system. In *IEEE Proc. Int. Conf. on Multimedia Computing and Systems*, pages 344–351, June 1997.

[82] Eamonn Keogh and Michael Pazzani. Relevance feedback retrieval of time series. In *22th ACM-SIGIR Conference*, 1999.

[83] Martin L. Kersten and M.F.N de Boer. Query optimization strategies for browsing sessions. In *IEEE 10th Int. Conf. on Data Engineering (ICDE)*, pages 478–487, February 1994.

[84] Won Kim. Unisql/x unified relational and object-oriented database system. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, page 481, 1994.

[85] Robert R. Korfhage. *Information Storage and Retrieval.* Wiley Computer Publishing, 1997.

[86] F. Korn, N. Sidiropoulos, and C. Faloutsos. Fast nearest neighbor search in medical image databases. *Proc. of VLDB*, 1996.

[87] Nick Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. 14th Int. Conf. on Data Engineering*, pages 466–475, 1998.

[88] Gerald Kowalski. *Information Retrieval Systems: theory and implementation.* Kluwer Academic Publishers, 1997.

[89] Amlan Kundu and Jia-Lin Chen. Texture Classification Using QMF Bank-Based Subband Decomposition. *CVGIP: Graphical Models and Image Processing*, 54(5):369–384, September 1992.

[90] Andrew Laine and Jian Fan. Texture Classification by Wavelet packet Signatures. *IEEE Trans. Patt. Recog. and Mach. Intell.*, 15(11):1186–1191, 1993.

[91] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system, September 1997.

[92] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34(63):50–63, October 1991.

[93] Ray R. Larson. Geographic information retrieval and spatial browsing. *http://sherlock.berkeley.edu/geo_ir/PART1.html*, 1997.

[94] William Leigh and Noemi Paz. The use of sql and second generation database management systems for data processing and information retrieval in libraries. *Information Technology and Libraries*, 8:400–407, December 1989.

[95] H. V. Lin, K.and Jagadish and C. Faloutsos. The TV-tree - an index stucture for high dimensional data. In *VLDB Journal*, 1994.

[96] Fang Liu and R.W. Picard. Periodicity, directionality, and randomness: Wold features for image modeling and retrieval. *IEEE Trans. Patt. Recog. and Mach. Intell.*, 18(7), July 1996.

[97] G. Lu and A. Sajjanhar. Region-based shape representation and similarity measure suitable for content-based image retrieval. *Springer Verlag Multimedia Systems*, 1999.

[98] Carol Lundquist et. al. A Parallel Relational Database Management System Approach to Relevance Feedback in Information Retrieval. *JASIS*, 50(5), 1999.

[99] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE T-PAMI special issue on Digital Libraries*, Nov. 1996.

[100] B.S. Manjunath and W.Y. Ma. Texture features for browsing and retrieval of image data. Technical report, CIPR TR-95-06, 1995.

[101] R. Mehrotra and J. Gary. Similar shape retrieval in shape data management. *IEEE Computer*, 1995.

[102] Sharad Mehrotra, Kaushik Chakrabarti, Michael Ortega, Yong Rui, and Thomas S. Huang. Towards Extending Information Retrieval Techniques for Multimedia Retrieval. In *3rd International Workshop on Multimedia Information Systems, Como, Italy*, 1997.

[103] T. P. Minka and R. W. Picard. Interactive Learning Using a "society of models". Technical Report 349, MIT Media Lab, 1996.

[104] Priti Mishra and Margareth H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1), March 1992.

[105] Makoto Miyahara et al. Mathematical Transform of (R,G,B) Color Data to Munsell (H,V,C) Color Data. In *SPIE Visual Communications and Image Processing'88*, volume 1001, 1988. 650.

[106] Amihai Motro. VAGUE: A user interface to relational databases that permits vague queries. *ACM TOIS*, 6(3):187–214, July 1988.

[107] Amihai Motro. FLEX: A tolerant and cooperative user interface databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–246, 1990.

[108] Apostol Natsev, Y. Chang, J. Smith, and J. Vitter. Supporting incremental join queries on ranked inputs. In *Very Large Databases (VLDB)*, pages 281–290, 2001.

[109] Surya Nepal and M.V. Ramakrishna. Query Processing Issues in Image (multimedia) Databases. In *IEEE ICDE*, 1999.

[110] NIST. Text retrieval conference. http://trec.nist.gov. Managed by the National Institute of Standards and Technology.

[111] Virginia E. Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer, Special Issue on Content-based Image Retrieval Systems*, Sept. 1995.

[112] ORACLE. *Oracle 8i Concepts, Release 8.1.6.* Oracle, December 1999.

[113] Michael Ortega, Kaushik Chakrabarti, Kriengkrai Porkaew, and Sharad Mehrotra. Cross media validation in a multimedia retrieval system. *ACM Digital Libraries 98 Workshop on Metrics in Digital Libraries*, 1998.

[114] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Sharad Mehrotra, and Thomas S. Huang. Supporting similarity queries in mars. In *Proceedings of the 5th ACM Int. Multimedia Conference*, pages 403–413, Seattle, Washington, November 1997.

[115] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Kriengkrai Porkaew, Sharad Mehrotra, and Thomas S. Huang. Supporting Ranked Boolean Similarity Queries in MARS. *IEEE Trans. Knowledge and Data Engineering*, 10(6):905–925, December 1998.

[116] Michael Ortega-Binderberger, Sharad Mehrotra, Kaushik Chakrabarti, and Kriengkrai Porkaew. Webmars: A multimedia search ending. In *Proceedings of the 2000 Multimedia Information System Workshop, Chicago, IL*, 2000.

[117] Apostolos N. Papadopoulos and Yannis Manolopoulos. Similarity query processing using disk arrays. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 225–236, 1998.

[118] Simon Parsons. Current approaches to handling imperfect information in data and knowledge bases. *IEEE Trans. Knowledge and Data Engineering*, 8(3):353–372, June 1996.

[119] A. Pentland, R.W. Picard, and S.Sclaroff. Photobook: Content-based manipulation of image databases. *International Journal of Computer Vision*, 1996.

[120] A. Pentland, R.W.Picard, and S. Sclaroff. Photobook: Tools for content-based manipulation of image databases. In *Proc SPIE*, volume 2185, pages 34–47, 1994.

[121] A. Pentland, R.W.Picard, and S.Sclaroff. Photobook: Tools for content-based manipulation of image databases. In *Proc. Storage and Retrieval for Image and Video Databases II*, volume 2, pages 34–47, Bellingham, Wash, 1994.

[122] K. Porkaew, K. Chakrabarti, and S. Mehrotra. Query refinement for content-based multimedia retrieval in MARS. *Proceedings of ACM Multimedia Conference*, 1999.

[123] Kriengkrai Porkaew, Sharad Mehrotra, and Michael Ortega. Query reformulation for content based multimedia retrieval in mars. In *IEEE International Conference on Multimedia Computing and Systems (ICMCS 99)*, June 1999.

[124] Kriengkrai Porkaew, Sharad Mehrotra, and Michael Ortega. Query reformulation for content based multimedia retrieval in mars. Technical Report TR-DB-99-03, University of California at Irvine, 1999.

[125] Kriengkrai Porkaew, Sharad Mehrotra, Michael Ortega, and Kaushik Chakrabarti. Similarity search using multiple examples in mars. In *Proc. Visual'99*, June 1999.

[126] András Prékopa, Béla Vizvári, Gábor Regős, and Linchun Gao. Bounding the probability of the union of events by the use of aggregation and disaggregation in linear programs. Technical Report RRR4-2001, Rutgers Center for Operations Research (Rutcor), Rutgers University, Jan. 2001.

[127] F. Rabitti and P. Stanchev. GRIM DBMS: A GRaphical IMage Database Management System. In *Visual Database Systems, IFIP TC2/WG2.6 Working Conference on Visual Database Systems*, pages 415–430, 1989.

[128] K. V. S. N. Raju and A. K. Majumdar. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *ACM Transactions on Database Systems*, 13(2), 1988.

[129] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD*, 1981.

[130] J.J. Rocchio. Relevance feedback in information retrieval. In Gerard Salton, editor, *The SMART Retrieval System*, pages 313–323. Prentice–Hall, Englewood NJ, 1971.

[131] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proceedings of SIGMOD*, 1995.

[132] Y. Rui, T. Huang, and S. Mehrotra. Relevance feedback techniques in interactive content-based image retrieval. *Proc. of IS&T and SPIE Storage and Retrieval of Image and Video Databases*, 1998.

[133] Yong Rui, Thomas S. Huang, Sharad Mehrotra, and Michael Ortega. Automatic matching tool selection using relevance feedback in MARS. In *Proc. of 2nd Int. Conf. on Visual Information Systems*, 1997.

[134] Yong Rui, Thomas S. Huang, Michael Ortega, and Sharad Mehrotra. Relevance feedback: A power tool for interactive content-based image retrieval. *IEEE Trans. Circuits and Systems for Video Technology*, 8(5):644–655, September 1998.

[135] Yong Rui, Alfred C. She, and Thomas S. Huang. Modified Fourier Descriptors for Shape Representation – A Practical Approach. In *Proceeding of First International Workshop on Image Databases and Multi Media Search*, 1996. Amsterdam, The Netherlands.

[136] G. Salton, Edward Fox, and E. Voorhees. Extended Boolean Information Retrieval. *Communications of the ACM*, 26(11):1022–1036, November 1983.

[137] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Computer Science Series, 1983.

[138] T. Seidl and H. Kriegel. Efficient user-adaptable similarity search in large multimedia databases. In *Very Large Databases (VLDB)*, 1997.

[139] T. Seidl and H. Kriegel. Optimal multistep k-nearest neighbor search. *Proc. of ACM SIGMOD*, 1998.

[140] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, Boston, MA, USA, May 30 - June 1 1979. ACM.

[141] Timos Sellis, N. Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, 1987.

[142] John C. Shafer and Rakesh Agrawal. Parallel algorithms for high-dimensional proximity joins for data mining applications. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 176–185, 1997.

[143] John C. Shafer and Rakesh Agrawal. Continuous querying in database-centric web applications. In *WWW9 conference*, Amsterdan, Netherlands, May 2000.

[144] W. M. Shaw. Term-relevance computations and perfect retrieval performance. *Information Processing and Management.*

[145] Chris Sherman and Gary Price. *The Invisible Web.* Cyber Ago Books, 2001.

[146] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-dimensional similarity joins. In *Proc. 13th Int. Conf. on Data Engineering*, pages 301–311, 1997.

[147] John R. Smith and Shih-Fu Chang. Querying by color regions using the visualseek content-based visual query system. Technical report, Columbia Univ.

[148] John R. Smith and Shih-Fu Chang. Transform Features for Texture Classification and Discrimination in Large Image Databases. In *Proc. IEEE Int. Conf. on Image Proc.*, 1994.

[149] John R. Smith and Shih-Fu Chang. Single Color Extraction and Image Query. In *Proc. IEEE Int. Conf. on Image Proc.*, 1995.

[150] John R. Smith and Shih-Fu Chang. Tools and techniques for color image retrieval. In *IS & T/SPIE proceedings Vol.2670, Storage & Retrieval for Image and Video Databases IV*, 1995.

[151] John R. Smith and Shih-Fu Chang. Automated Binary Texture Feature Sets for Image Retrieval. In *Proc ICASSP-96*, Atlanta, GA, 1996.

[152] T. G. Aguierre Smith. Parsing movies in context. In *Summer Usenix Conference, Nashville, Tennessee*, pages 157–167, 1991.

[153] Gabriele Sonnenberger. Exploiting the functionality of object-oriented database management systems for information retrieval. *Bulletin of IEEE Computer Society Technical Committee on Data Engineering*, 1996.

[154] Amanda Spink and Tefko Saracevic. Human—computer interaction in information retrieval: nature and manifestations of feedback. *Interacting with Computers*, 10(3):249–267, 1998.

[155] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, 1991.

[156] Michael Stonebreaker and Dorothy Moore. *Object-Relational DBMSs, The Next Great Wave.* Morgan Kaufman, 1996.

[157] Markus Stricker and Markus Orengo. Similarity of Color Images. In *Proc. SPIE Conf. on Vis. Commun. and Image Proc.*, 1995.

[158] Michael Swain and Dana Ballard. Color Indexing. *International Journal of Computer Vision*, 7(1), 1991.

[159] Sybase. *Sybase Adaptive Server Enterprise, Version 12.0.* Sybase, 1999.

[160] Hideyuki Tamura et al. Texture Features Corresponding to Visual Perception. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-8(6), June 1978.

[161] Hideyuki Tamura, Shunji Mori, and Takashi Yamawaki. Texture Features Corresponding to Visual Perception. *IEEE Trans. on Sys, Man, and Cyb*, SMC-8(6), 1978.

[162] K. S. Thyagarajan, Tom Nguyen, and Charles Persons. A Maximum Likelihood approach to Texture Classification Using Wavelet Transform. In *Proc. IEEE Int. Conf. on Image Proc.*, 1994.

[163] TPC. Transaction processing performance council (tpc). http://www.tpc.org.

[164] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 1991.

[165] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[166] S. R. Vasanthakumar, James P. Callan, and Bruce W. Croft. Integrating inquery with an rdbms to support text retrieval. *Bulletin of IEEE Computer Society Technical Committee on Data Engineering*, 1996.

[167] D. White and R. Jain. Similarity indexing with the ss-tree. *Proc. of ICDE*, 1995.

[168] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compression and Indexing Documents and Images.* Morgan Kaufmann Publishers, May 1999.

[169] L. Wu, C. Faloutsos, K. Sycara, and T. Payne. FALCON: Feedback adaptive loop for content-based retrieval. *Proceedings of VLDB Conference*, 2000.

[170] Lofti A. Zadeh. Fuzzy logic. *IEEE Computer*, 21(4):83–93, 1988.

[171] Lofti A. Zadeh. Knowledge representation in fuzzy logic. *IEEE Trans. Knowledge and Data Engineering*, 1(1):89–100, 1989.

[172] Carlo Zaniolo, Stefano Ceri, Christos Faoutsos, Richard T. Snordgrass, V.S. Subrahmanian, and Roberto Zicardi. *Advanced Database Systems.* Morgan Kaufman, 1997.

Vita

Michael Ortega Binderberger received his B.Eng. degree in Computer Engineering from the Instituto Tecnológico Autónomo de México (ITAM), Mexico in 1994, and a M.S. degree in Computer Science from the University of Illinois at Urbana Champaign in 1999. He is currently finishing his Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign and is a visiting researcher at the University of California, Irvine. His research interests include multimedia databases, information retrieval, decision support systems, data mining, highly available and reliable computing, and information security. He is the recipient of a Fulbright scholarship and an IBM fellowship. He is a member of the ACM (SIGMOD, SIGIR, and SIGKDD special interest groups), IEEE, and IEEE Computer Society. He was elected into the Honor Society of Phi Kappa Phi in 1998 for having perfect GPA (4.0/4.0) in graduate school.